Chapter 5
Graph rewriting systems

Graph rewriting systems (GRSs) (introduced in Barendregt *et al.* (1987a, b) are an extension of term rewriting systems where the terms are replaced by directed graphs. The main goal is to obtain a sound extension of the TRS model in which duplication of work is avoided via sharing of subgraphs (see Section 5.1). More information on this approach and on related approaches can be found in Ehrig *et al.* (1973), Raoult (1984), Staples (1980) and Kennaway (1990).

A computation in a GRS is specified by a set of graph rewrite rules that are used to rewrite a given initial graph to its final result. The rules contain graph patterns that can match some part of the graph. If the graph matches a rule it can be rewritten according to the specification in that rule. This specification makes it possible to construct an additional graph structure which is linked to the existing graph by redirecting all arcs from the root of the redex to the root of the result (Sections 5.2 and 5.3).

Many notions in GRSs are similar to the notions in TRSs and also many properties of subclasses carry over from TRSs to GRSs (Sections 5.4, 5.6 and 5.7). For instance, GRSs are also in general not confluent. However, similar to the TRS world it is possible to define confluent subclasses with normalizing strategies. As with term rewriting, priority can be added to graph rewrite rules. The functional strategy can be used to rewrite graphs in a GRS with priorities (Section 5.8).

A TRS can be converted (lifted) to a GRS (Section 5.8). This conversion has been made very easy by a special shorthand notation that is defined for GRSs (Section 5.5). In most cases a GRS obtained via lifting is syntactically equivalent to the original TRS. The GRS semantics is then applied instead of the TRS semantics. Rewriting graphs instead of terms has the advantage that duplication of work is avoided. Generally, fewer rewrite steps are needed to reach a normal form. The resulting graph in normal form can be converted back to a term again (unravelled). Under certain conditions the same reduct is obtained via this *term graph rewriting* as would have been the case when the original term was reduced in the term world. In particular, sharing can be used safely in lifted FTRSs.

Graph rewriting can be generalized to allow multiple redirections (Section 5.9). Such a generalized graph rewriting model allows much more complex changes of a graph structure than is necessary for lifted TRSs. Generalized graph rewriting can model basic concepts of other languages than pure functional languages, such as those found in logic languages and even in imperative languages. Generalized GRSs are used for instance to investigate the possibility of combining different kinds of language concepts into one framework.

GRSs are referentially transparent, since in a rewrite step all arcs to the root of the subgraph matching a pattern are redirected to the result. GRSs with priority and the functional strategy very much resemble a functional language and therefore can serve very well as a basis for functional languages (Section 5.10). Transformation schemes that specify the conversion of a functional program into an equivalent GRS are discussed in detail in Part 4. An efficient implementation of a generalized GRS is very hard to obtain. However, efficient implementations of functional language-like GRSs are very possible (Section 5.11). In Part 4, efficient implementations of functional languages based on GRSs are treated in detail.

5.1 Extending TRSs to GRSs

In this section the differences and correspondences between *term* and *graph* rewriting are briefly explained in order to give the reader a first intuition on the subject.

TRS terms can be regarded as trees (see Chapter 4). The basic motivation behind the extension of TRSs to GRSs is to include the notion of *sharing of computation*. For this purpose the objects of rewriting become directed acyclic graphs instead of trees. The GRS model is even more general: it also allows cyclic structures to be manipulated. In TRSs rewriting of terms consists of replacing a subterm by another subterm. In graph rewriting the natural extension of sub*term* replacement is sub*graph* replacement. This subgraph replacement is achieved via a change of the graph structure retaining as much sharing as possible.

5.1.1 Terms versus graphs

There are many similarities between the tree representation of terms and the graphs as used in GRSs. However, in order to be able to express sharing and cycles a unique *label* (like an address) is added to every node, called the **node identifier** or **node-id** (Figure 5.1). A node is then referred to by its node-id. A subgraph can be identified via the node-id of the root of that subgraph.

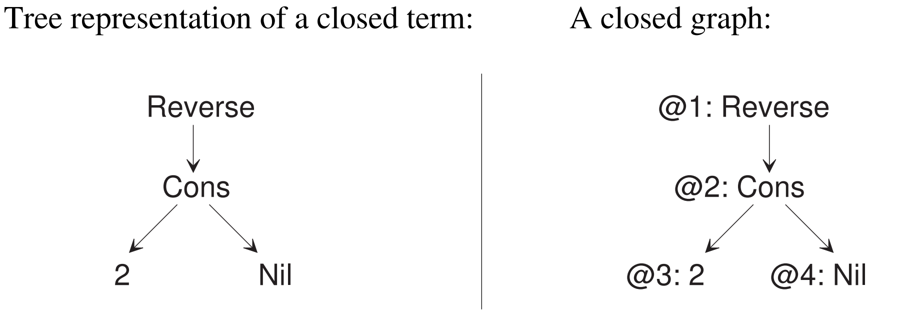


Figure 5.1 Closed terms in TRSs versus closed graphs in GRSs.

5.1.2 Term rewrite rules versus graph rewrite rules

In a *graph rewrite rule* graphs are specified instead of terms.

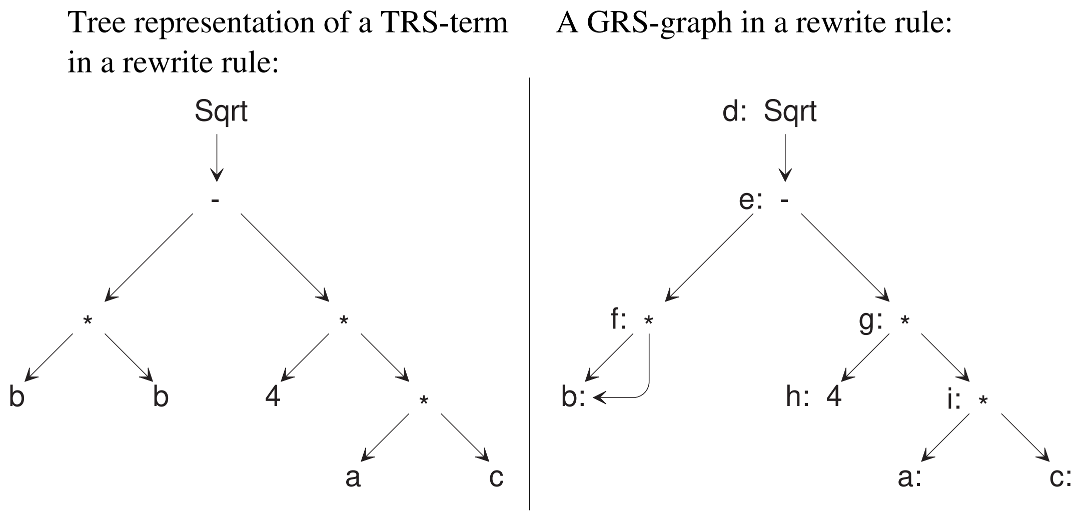


Figure 5.2 Terms versus graphs in rewrite rules.

The graphs that are specified in the rewrite rules of a GRS are **open graphs**, i.e. these graphs are labelled with node-id *variables*. The graph that is rewritten is a **closed graph**, i.e. the nodes of the graph are labelled with node-id *constants*. There is a difference between variables in TRSs and variables in GRSs. Variables that appear in the rewrite rules of a TRS are bound to a whole subterm (or subtree) of the term to be rewritten. Since a subgraph of a graph to be rewritten can be identified by the node-id of the root node of that subgraph, a node-id variable in a GRS-rule is bound to just a simple node-id constant.

5.1.3 Term rewriting versus graph rewriting

An important difference between term rewriting and graph rewriting is that with term rewriting subterms are duplicated if a corresponding variable appears more than once on the right-hand side whereas with graph rewriting the subterms will be shared. This is caused by the fact that the variables in graph rewriting rules stand for node-ids.

Consider the following TRS:

```
Double x               →    + x x
Heavy_Computation      →    Easy_Number
```

The term Double Heavy_Computation will be reduced in this TRS as follows (using a lazy strategy):

```
Double Heavy_Computation
→Double                 + Heavy_Computation Heavy_Computation
→Heavy_Computation      + Easy_Number Heavy_Computation
→Heavy_Computation      + Easy_Number Easy_Number
```

which is displayed in Figure 5.3. using the tree representation of TRS terms.

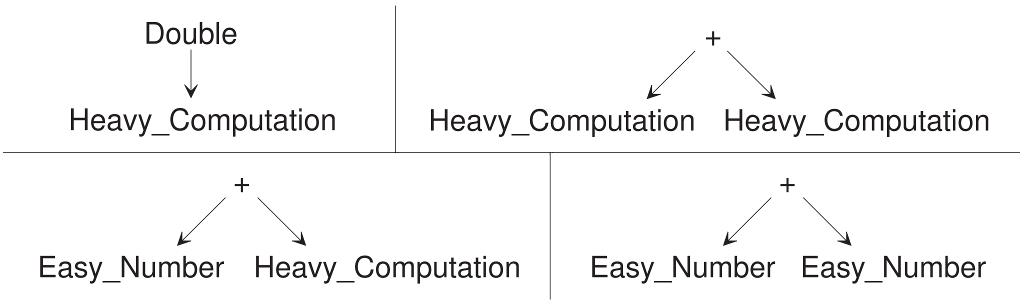


Figure 5.3 Four snapshots of the term rewriting process.

The example above shows that in term rewriting work may be duplicated by subtree replacement. In GRSs duplication of work is avoided. The graph structure makes it possible to share a subgraph representing a computation that has to be performed (see also Section 5.8). If such a subgraph is replaced by the subgraph representing the result of the computation all arcs pointing to the computation automatically become arcs pointing to that result. Subgraph replacement in GRSs is accomplished via redirection of arcs in the graph.

Avoiding duplication of work by sharing computations.

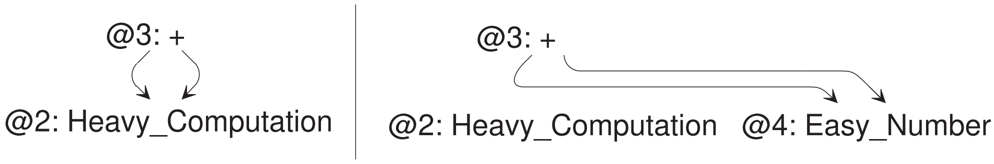


Figure 5.4 Two snapshots of the graph rewriting process.

In Figure 5.4 the same rules as above are used with graph rewriting semantics instead of term rewriting semantics. The snapshots in Figure 5.4 show that the right-hand side of the Double rule induced the sharing of the expression representing the heavy computation. The reduction of the heavy computation to the easy number is only done once after which all arcs to the root of the heavy computation subgraph are redirected to the root of the result (2nd snapshot).

5.2 GRSs

A GRS consists of a collection of rewrite rules. These rewrite rules specify how a given *graph* can be *rewritten* (*reduced*) to another graph.

In a GRS there are two syntactic ways of specifying graphs and rules: the shorthand form and the canonical form. In the *shorthand form* the graph structure need not be fully specified if it can be deduced from the notation. The shorthand notation is designed in such a way that those GRSs that very much resemble TRSs can also be written in a syntactic form that is very much like a TRS. In the *canonical form* everything is specified explicitly. Generally, the shorthand form is used for specifying computations whereas the canonical form is mainly used to analyse the semantics of GRSs. Keep in mind that both forms denote exactly the same rules and objects, so they can be interchanged freely.

In this section the canonical form will be treated. This form will also be used in explaining the semantics in Section 5.3. In Section 5.5 the more readable TRS-like shorthand form will be explained.

5.2.1 Graphs

A collection of **graphs** over a set of *constant symbols* and a set of *node identifiers* (or *node-ids*) is defined by the following syntactic definition:

```
Graph       =  NodeDef {',' NodeDef} ;
NodeDef     =  Nodeid ':' Node ;
Node        =  Symbol {Arg}     |  EmptyNode ;
Symbol      =  Constant ;
Arg         =  Nodeid ;
Nodeid      =  NodeidVariable   |  NodeidConstant ;
EmptyNode   =  '⊥' ;
```

Constants always begin with an upper case character or some special character (a digit, an asterisk etc.), node-id variables with a lower case character, and node-id constants with the character @ (the *at* sign). An empty node contains a special symbol. The role of empty nodes is discussed in Section 5.6.

Some graphs in canonical form:

```
@1:  Hd @2,
@2:  Cons @3 @4,
@3:  0,
@4:  Nil
```

```
@1:  Add @2 @2,
@2:  Fac @3,
@3:  1000
```

```
@1:  Cons @2 @1
@2:  1
```

```
@1:  Tuple @2 @3 @4 @5 @6,
@2:  1,
@3:  –3,
@4:  5,
@5:  –7,
@6:  11
```

```
x:  Add y z,
y:  Succ z,
z:  Zero
```

Their pictorial equivalent:

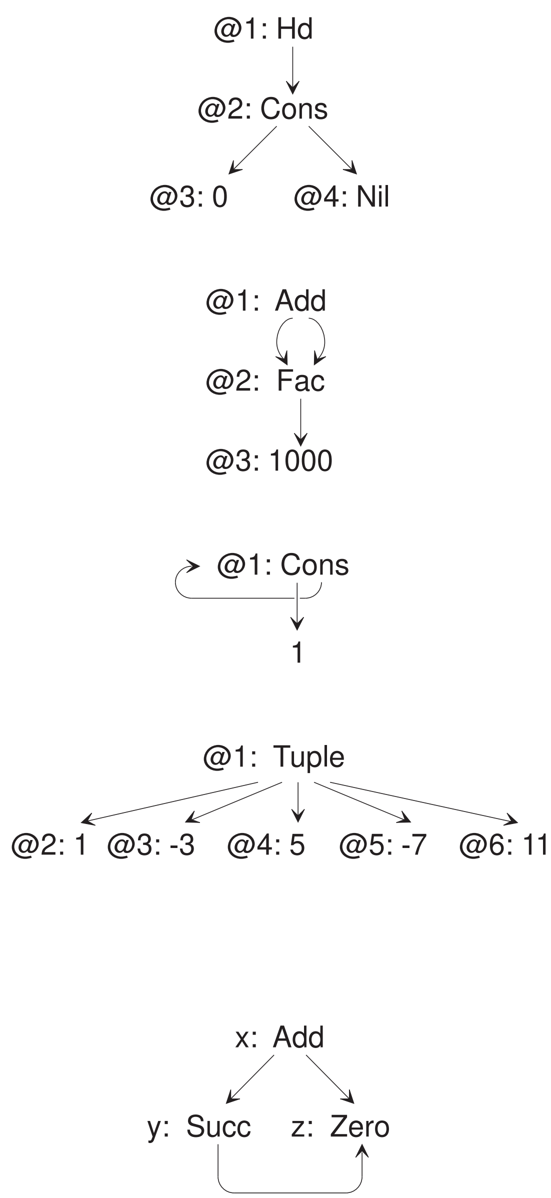


Figure 5.5 Some graphs.

A **graph** consists of a set of **node definitions**. Each node in the graph has a unique **node-identifier** or **node-id** associated with it (Figure 5.5). Occurrences of node-ids before a colon are **defining occurrences**. Each **node** consists of a **constant symbol** with a (possibly empty) sequence of **arguments**: node-ids defining *arcs* pointing to (other) nodes in the graph. There are **node-id constants** and **node-id variables.** A graph that contains no node-id variables is called a **closed graph**, otherwise it is called an **open graph.** A **subgraph** of a graph is a set of node definitions contained in the given graph. A graph can be cyclic. Symbols have fixed arities.

There is a distinguished node in every graph that is the *root* of that graph. When the set of node-id definitions is textually enumerated, the topmost node-id definition is always the definition of the **root** of the graph. A **path** in a graph is a sequence of node-ids of the graph such that each node-id is an argument of the node associated with its predecessor in the sequence; the path is said to be a **path from** *A* **to** *B* where *A* and *B* are the nodes associated with respectively the first and the last node-id of the path. A node *A* is **reachable** from a node *B* if there exists a path from *B* to *A*. When all nodes in the graph are reachable from the root of the graph, the graph is said to be a **connected graph**. In GRSs all graphs (also in the rules) are connected. **The subgraph of a node-id** *n* in a graph *g* is defined as the connected subgraph of *g* which is rooted at *n* and contains the node definitions of all nodes in *g* that are reachable from *n*.

Two closed graphs are considered to be **equivalent** when the graphs are identical up to the node-id constants. This means that the graphs have the same graph structure and that corresponding nodes contain the same symbol with the same arity. So between two equivalent graphs there exists a rooted isomorphism. From now on graphs will be considered to be indistinguishable with respect to this equivalency.

From a standard graph theory point of view, a GRS graph is a *connected directed graph* in which each node is labelled with a symbol whose set of out-arcs is given an ordering. In standard graph theory, cycles in directed graphs are usually referred to as *circuits*.

5.2.2 Graph rewriting systems

A **graph rewriting system** (GRS) is a collection of *rewrite rules* $\alpha_i \rightarrow \beta_i, \varphi_i$ over sets of constant symbols and node-id variables obeying the syntactical restrictions described below. The object that is manipulated in a GRS is a connected closed graph called the **data graph.** When there is no confusion, the data graph is simply called the **graph**.

Syntax of GRSs (canonical form)

In the notation below the square brackets specify an optional occurrence of the enclosed objects:

```
CanonicalGRS     =  {Rule} ;
Rule             =  RedexPattern '→' [ContractumPattern ','] Redirection ;
RedexPattern     =  Graph ;
```

```
ContractumPattern  =  Graph ;
Redirection        =  Nodeid ':=' Nodeid ;
```

The left-hand side α_i of a rewrite rule consists of a connected open graph that is called the **redex pattern.** The right-hand side β_i consists of a connected open graph called the **contractum pattern** (optional) and a **redirection** φ_i.

The patterns are *open* graphs not containing node-id constants (only node-id *variables*). Every node-id variable must have at most one defining occurrence within a rule. A node-id variable that appears on the right-hand side of a rule, must either have a defining occurrence on this right-hand side or it must appear on the corresponding left-hand side. Note that node-id variables are allowed only in rewrite rules whereas node-id constants are allowed only in the data graph.

The left node-id of the redirection is always the root of the redex pattern. The right node-id of the redirection must either be the root of the contractum pattern or a node-id that occurs in the redex pattern. The redirection specifies to which node arcs to the root of the redex pattern are redirected.

The first symbol in a redex pattern is called the **function symbol**. Function symbols can also occur at positions that are not the head of the pattern. A symbol that does not occur at the head of any pattern in the GRS is called a **constructor symbol**.

Example of a GRS (compare this with the example in Section 5.5 that uses the much more readable shorthand notation):

```
r: Hd x,
x: Cons a b   →   r := a

r: Fac x,
x: 0          →   r':  1,      r := r'

r: Fac n      →   r':  * n u,
                  u:  Fac v,
                  v:  – n w,
                  w: 1,      r := r'

r: Start      →   r':  Fac a,
                  a:  Hd b,
                  b:  Cons c d,
                  c:  1000,
                  d:  Nil,    r: = r'
```

5.3 Rewriting with a GRS

In this section the full operational semantics for GRSs is given. As said before, a GRS is used to rewrite a given closed data graph γ. The data graph that is to be rewritten can be compared with the initial expression to be evaluated as present in TRSs and in Miranda.

5.3.1 The initial graph

One can specify on which data graph the set of rules has to be applied. However, since change of graph structure is obtained via redirecting existing arcs, it is more convenient for the specification of the semantics to assume a fixed initial graph to be present. This initial graph consists of the root node that always remains the root of the data graph, even after rewriting. The purpose of the root node is always to have an arc pointing to the actual graph to be rewritten. Initially the root node contains an arc to the start node. A GRS should contain a rule matching the start node to initiate the computation.

The initial graph in linear notation:	In pictorial notation:
@DataRoot: Graph @StartNode, @StartNode: Start	@DataRoot: Graph ↓ @StartNode: Start

Requiring an initial graph does not impose a serious restriction because on the right-hand side of a start rule one can always specify the desired graph one would actually like to rewrite. When such a start rule is applied, the arc pointing from the root node to the start node will be redirected to this desired graph. This graph then becomes the object of rewriting. For reasons of simplicity sometimes the start rule is omitted in examples and a desired graph is specified directly.

5.3.2 Redexes

As explained above, a redex pattern consists of constant symbols and node-id variables. An **instance** of a redex pattern is a subgraph of the data graph, such that there is a mapping from the pattern to this subgraph. Such a mapping must be the identity on symbols and it has to preserve the node structure. The corresponding subgraph is called a **redex** (*red*ucible *ex*pression). The redex pattern (and the corresponding rewrite rule) is said to **match** the redex. As in TRSs, it depends on the rules present in the GRS whether or not a subgraph is a redex or not.

In order to define a match more formally, some terminology has to be introduced. Let F be a set of symbols and N be a set of node-ids. Further, let C be a function (the *contents function*) from N to $F \times N^*$.

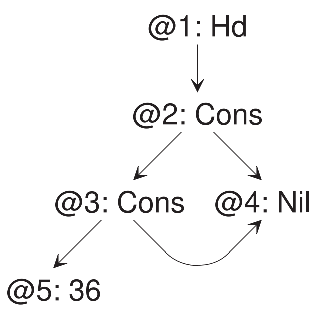
Then C specifies a GRS graph over F and N. Furthermore, for each GRS graph γ there is a unique contents function: C_γ. The canonical form can be regarded as a tabulation of the contents function. Let γ be a GRS graph, and ρ the set of rewrite rules. Then μ is a mapping from the node-id variables in a redex pattern π of ρ to node-id constants of the graph γ such that for every node with node-id x in π with contents, say $C_\pi(\mathrm{x}) = \mathrm{S}\ \mathrm{x}_1\ \mathrm{x}_2 \ldots \mathrm{x}_n$ it holds that $C_\gamma(\mu(\mathrm{x})) = \mathrm{S}\ \mu(\mathrm{x}_1)\ \mu(\mathrm{x}_2) \ldots \mu(\mathrm{x}_n)$. That is, μ preserves the node structure.

Consider the following GRS:

```
r:  Hd x,
x:  Cons a b       →             r := a
```

and the following data graph:

```
@1:  Hd @2,
@2:  Cons @3 @4,
@3:  Cons @5 @4,
@4:  Nil,
@5:  36
```



The mapping μ is defined by: μ(r) = @1, μ(x) = @2, μ(a) = @3, μ(b) = @4.

5.3.3 Rewriting a graph

The semantics of a GRS is explained using the canonical form. The operational semantics (Barendregt *et al.*, 1987b) is similar to Staples' model of graph rewriting (Staples, 1980). A categorical semantics of graph rewriting is given in Kennaway (1990).

A **rewrite** (or **reduction**) of a graph γ and the set of rewrite rules ρ is performed in a number of steps. Consider a rewrite rule and a redex that matches the rule with a corresponding concrete mapping μ. The basic idea is to construct a new data graph out of the old data graph and out of a graph called the **contractum** via redirection of arcs. The contractum is only constructed if a contractum pattern is specified on the right-hand side of the chosen rule. More precisely, the operational semantics of graph rewriting is defined by the following steps:

(1) Choose a redex together with a matching rule and a mapping μ.

(2) When a contractum pattern is specified in the matching rule the contractum is constructed in the following way:

- invent new constant node-ids (not present in γ) for each node-id variable on a defining occurrence in the contractum pattern. This mapping is called μ'. These new node-ids correspond to the identifiers of new nodes to be introduced during rewriting.
- apply μ'', the combination of μ and μ', to the node-ids in the contractum pattern of the rule yielding the contractum graph γ'.

If no contractum pattern is specified, then γ' is empty.

(3) A graph γ'' is constructed by taking the union of γ and γ'.

(4) The redirection in a rule takes the form `r := n` where `r` is the root node of the redex pattern. Now all arcs pointing to the root node of the redex are redirected to the specified node-id. By applying μ'', node-id constants `@R` and `@N` are determined from the node-id variables that occur in the redirection. In terms of the syntactic representation, redirection is performed by substituting `@N` for every occurrence of `@R` *as an argument* in every node of the graph γ''. This results in the new graph γ'''. Note that the node *definition* of `@R` still remains.

(5) The graph γ''' thus constructed may be a graph that is disconnected. In order to make the resulting new data graph connected again only those nodes are considered to be part of the new data graph that are reachable from `@DataRoot`, the root of the initial graph. All nodes that are not reachable from the root of the graph are removed from the graph. This is known as **garbage collection**.

Since the node that is the root of the redex plays such an important role in the rewrite, one sometimes by abuse of language speaks of **rewriting** or **reducing a node** when one means rewriting the *graph* of which the node is the root.

5.3.4 A small example

When the rewriting process starts, the data graph γ always consists of the following initial graph:

```
@DataRoot :  Graph @StartNode,
@StartNode:  Start
```

```
@DataRoot: Graph
     |
     v
@StartNode: Start
```

In the following example the data graph will be rewritten until it contains no redexes (to *normal form*) given the following graph rewrite rules.

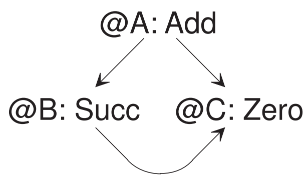
```
x: Add y z,
y: Zero     →   x := z                                    (1)

x: Add y z,
y: Succ a   →   m:   Succ n,
                n:   Add a z,
                x := m                                    (2)

x: Start    →   m:   Add n o,
                n:   Succ o,
                o:   Zero,
                x := m                                    (3)
```

The following semantic actions are performed:

(1) The start node is the only redex matching rule (3). The mapping is trivial: μ(x) = @StartNode.

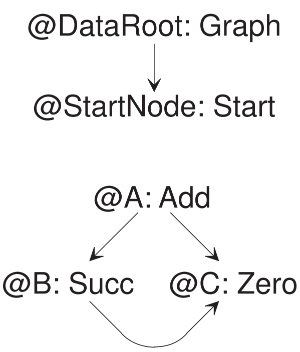
(2) Invent new node-ids for the node-id variables defined in the contractum pattern: μ'(m) = @A, μ'(n) = @B, μ'(o) = @C. Applying μ'' (the combination of μ and μ') to the node-ids in the contractum pattern will leave this unchanged as x does not appear in it. Now the contractum γ' can be constructed:

```
@A:  Add @B @C,
@B:  Succ @C,
@C:  Zero
```



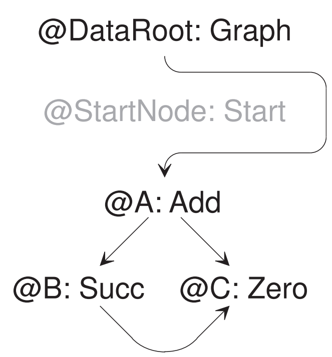
(3) The union of contractum γ' and graph γ is the new graph γ'':

```
@DataRoot : graph @StartNode,
@StartNode: Start,
@A:  Add @B @C,
@B:  Succ @C,
@C:  Zero
```



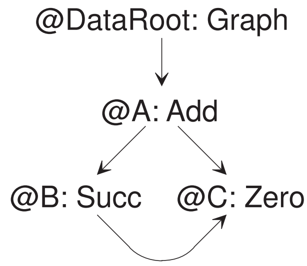
(4) μ''(x) = @StartNode has to be redirected to μ''(m) = @A. All occurrences of @StartNode as an argument will be replaced by occurrences of @A. The graph γ''' resulting after rewriting is now:

```
@DataRoot : Graph @A,
@StartNode: Start,
@A: Add @B @C,
@B: Succ @C,
@C: Zero
```



(5) The start node is no longer reachable from the data root node: it is garbage. Hence it is removed:

```
@DataRoot:   Graph @A,
@A: Add @B @C,
@B: Succ @C,
@C: Zero
```



This completes one rewrite. The graph still contains a redex. So reduction can go on:

(1) There is only one redex to choose.
It matches rule 2: μ(x) = @A, μ(y) = @B, μ(z) = @C, μ(a) = @C.

(2) Invent new node-ids and map the variables as follows: μ'(m) = @D, μ'(n) = @E. The contractum can now be constructed:

```
@D: Succ @E,
@E: Add @C @C
```

(3) The union of γ' and γ is γ'':

```
@DataRoot:   Graph @A,
@A: Add @B @C,
@B: Succ @C,
@C: Zero,
@D: Succ @E,
@E: Add @C @C
```

(4) μ''(x) = @A is redirected to μ''(m) = @D:

```
@DataRoot:   Graph @D,
@A: Add @B @C,
```

```
@B:  Succ @C,
@C:  Zero,
@D:  Succ @E,
@E:  Add @C @C
```

(5) The graph γ''' after removing garbage is:

```
@DataRoot:   Graph @D,
@C:  Zero,
@D:  Succ @E,
@E:  Add @C @C
```

It is now clear how this process may continue: @E is a redex and it matches rule 1: μ(x) = @E, μ(y) = @C, μ(z) = @C. The strategy chooses this redex; there is no new contractum graph but just a single redirection that takes μ''(x) = @E to μ''(z) = @C, yielding the expected normal form:

```
@DataRoot:   Graph @D,
@C:  Zero,
@D:  Succ @C
```

5.3.5 δ-rules

δ-rules are used in the same way as in TRSs and λ-calculus.

5.4 Reduction sequences and normal forms

The notions defined in this section are all very similar to the corresponding notions in TRSs with the exception that computation is initiated by the start rule.

One reduction is often called a **reduction step** (or **rewrite step**). A **reduction sequence** (or **rewrite sequence**) of a graph is a sequence of zero or more reduction steps (or rewrites) performed on that graph. If graph γ_1 can be reduced by a sequence of reductions to graph γ_2, γ_2 is called a **reduct** of γ_1. This is denoted as: $\gamma_1 \rightarrow_* \gamma_2$. So $\rightarrow_*$ is the reflexive transitive closure of $\rightarrow$.

Given a GRS, a graph is in **normal form** if no rule in the GRS matches the graph or one of its subgraphs.

The root normal form property in GRSs is the natural generalization of the head normal form property in TRSs: a graph is in **root normal form** if the graph as a whole is not a redex and the graph as a whole will never become a redex. In general it is undecidable whether a graph is in root normal form.

A pattern **partially matches** a graph if firstly the symbol of the root of the pattern equals the symbol of the root of the graph and secondly in positions where symbols in the pattern are not syntactically equal to symbols in the graph, the corresponding subgraph is a redex or the subgraph itself is partially matching a rule. A graph is in **strong root normal form** if the graph does not partially match any rule. It is decidable whether or not a graph is in strong root normal form. A graph in strong root normal form does not partially match any rule, so it is also in root normal form. A graph in root normal form need not be in strong root normal form (see also the previous chapter). In the graph world the notions partial match and strong root normal form are only partially defined. The cases where these notions are not defined are cases of cycle-in-spine errors (see also Section 5.8).

Consider the following GRS:

```
A 0       →    1
```

For the term @N: A @N the notion partial match is undefined.

5.5 Shorthand form

Before some important properties and aspects of GRSs are discussed, first the shorthand notation is introduced. In the canonical form every node has a definition and definitions are not nested. Redirection is explicitly specified. In the **shorthand form** it is not necessary to specify all the node-ids. Furthermore redirection need not be specified explicitly. This notation makes it possible to use a more readable TRS-like notation for GRSs. The shorthand notation will be used throughout the rest of this chapter.

Syntax of GRS graphs and GRSs (shorthand form)

```
GRS                = {Rule} ;
Rule               = RedexPattern '→' ContractumPattern [',' Redirection]
                   | RedexPattern '→' Redirection ;
RedexPattern       = Graph ;
ContractumPattern  = Graph ;
Graph              = [Nodeid ':'] Node {',' NodeDef} ;
Nodeid             = NodeIdVariable     | NodeIdConstant ;
Node               = Symbol {Arg}       | EmptyNode ;
Symbol             = Constant ;
Arg                = Nodeid             | [Nodeid ':'] Symbol
                   | [Nodeid ':'] '(' Node ')' ;
EmptyNode          = ⊥ ;
NodeDef            = Nodeid ':' Node ;
Redirection        = Nodeid ':=' Nodeid  | Nodeid ;
```

Some graphs in shorthand form (compare this with the equivalent examples of Section 5.2 that use the canonical form):

```
Hd (Cons a b)    →    a

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

Ones     →    n: Cons 1 n

Start    →    Fac (Hd (Cons 1000 Nil))
```

The shorthand notation is more readable and hence generally this form is used for specifying computations. Both forms denote the same rules and objects so there is no semantic difference between them. The shorthand form is not only used in the rules, but also to denote graphs that are being reduced:

Consider the following example in shorthand form:

```
Add Zero z        →    z
Add (Succ a) z    →    Succ (Add a z)
```

and the following data graph:

```
Add (Succ @1) @1, @1: Zero
```

The data graph can be rewritten in the following way:

<u>Add (Succ @1) @1</u>, @1: Zero
$\rightarrow_{\text{Add}_2}$ Succ (<u>Add @1 @1</u>), @1: Zero
$\rightarrow_{\text{Add}_1}$ Succ Zero

The reduction of this graph is also given in canonical form in Section 5.3.

Translating shorthand form to canonical form

The shorthand form can be translated into canonical form via two steps:

(1) *Add explicit node-ids and flatten*
In the canonical form all nodes have explicit node-ids and there are no nested node definitions. In each rule for every argument that does not consist of a single node-id, a new node is defined with the same contents as that argument. The new node is labelled with a new unique node-id. The original argument is replaced by this new node-id. With this replacement brackets disappear. Adding explicit

node definitions and replacing corresponding arguments is repeated until there is no nesting and all node-ids are explicit.

The rules that were used before in the small example to illustrate the semantics of GRSs, denoted in the shorthand form:

```
Add Zero z        →   z                              (1)
Add (Succ a) z    →   Succ (Add a z)                 (2)

Start             →   Add (Succ o) o,
                      o: Zero                        (3)
```

Note the sharing on the right-hand side of the Start rule. The first transformation step towards canonical form replaces the rules by:

```
x: Add y z,
y: Zero           →   z                              (1')

x: Add y z,
y: Succ a         →   m: Succ n,
                      n: Add a z                     (2')

x: Start          →   m: Add n o,
                      n: Succ o,
                      o: Zero                        (3')
```

(2) *Make redirection explicit*

The root of the redex pattern is redirected to the root of the contractum pattern or if the right-hand side consists of a single node-id variable, it is redirected to that node-id variable.

The same example with explicit redirections in the canonical form:

```
x: Add y z,
y: Zero           →   x := z                         (1'')

x: Add y z,
y: Succ a         →   m: Succ n,
                      n: Add a z,
                      x := m                         (2'')

x: Start          →   m: Add n o,
                      n: Succ o,
                      o: Zero,
                      x := m                         (3'')
```

5.6 Properties of GRSs

Although there are many notions in GRSs that are similar to the notions in TRSs and also many properties of subclasses carry over from TRSs to GRSs (Smetsers, 1993), there are also subtle differences between these two worlds. These similarities and differences are discussed briefly in this section.

5.6.1 Sources of non-confluency

A GRS is **confluent** (or has the **Church–Rosser property**) iff for every two 'divergent' reduction sequences $\gamma \to_* \gamma_1$ and $\gamma \to_* \gamma_2$ there are two 'convergent' reduction sequences $\gamma_1 \to_* \gamma_3$ and $\gamma_2 \to_* \gamma_3$ (similar to the definition in TRS and the λ-calculus). The Church–Rosser property implies the *unique normal form property*.

Ambiguity

Just like in a TRS (partial) overlapping patterns in a GRS give rise to ambiguity and they can be a source of non-confluency. However, cyclic structures make it also possible that a subgraph can be matched in several ways on the same partially overlapping rule.

Example of several ways of matching with a cyclic structure:

```
r:  F x,
x:  F a    →     r := a
```

Take the following graph:

```
@1: F @2,
@2: F @1
```

This graph matches the rule in two ways. There are two possible mappings: μ(r) = @1, μ(x) = @2, μ(a) = @1 but also μ(r) = @2, μ(x) = @1, μ(a) = @2.

Comparing is not a source of non-confluency

In a TRS multiple occurrences of a variable in the left-hand side mean that the arguments are tested on their syntactic equality. In a GRS however, this means that a test is done on the equality of the node-ids of the arguments. If the node-ids are the same, the arguments are also syntactically equal. But subgraphs with the same contents need not have the same node-id of course. We conjecture that because of the different semantics, in a GRS comparing is *not* a source of non-confluency.

Consider the following example as a GRS (see also Section 4.4.3):

```
D x x   →   E
C x     →   D x (C x)
A       →   C A
```

In the rule for C on the right-hand side for the second argument of D a new node-id is invented for the node C x. This new node-id cannot be the same as the node-id corresponding with x. Since there is no rule C x → x these node-ids will remain different such that the rule for D can never be applied.

Self-embedding redexes

In GRSs a very special source of non-confluency exists, called *self-embedding redexes*. A redex is **self-embedding** iff there is a mapping μ for a rule ρ with redirection φ such that the corresponding μ'' applied to the redirection maps both left- and right-hand sides of the redirection to one and the same node-id constant in the graph. In other words: the root of the redex has to be redirected to itself. So such a self-embedding redex reduces to itself.

Consider the following GRS:

```
I  x  →   x
```

The graph @1: I @1 is a self-embedding redex. It will be reduced to itself as follows with the semantics as described up to now:

@1: I @1 $\to_{I}$ @1: I @1 $\to_{I}$ …

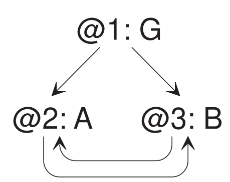
The example below shows that with self-embedding redexes a non-confluent situation can occur.

Self-embedding redexes can give rise to a non-confluent situation:

```
A  x  →   x
B  x  →   x
```

The graph

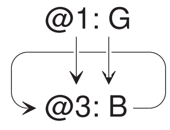
@1: G @2 @3,
@2: A @3,
@3: B @2



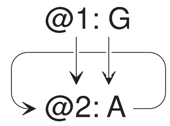
contains two redexes.

Depending on the order of evaluation two different self-embedding redexes are created:

@1: G @2 @3, @2: A @3, @3: B @2
$\rightarrow_A$ @1: G @3 @3, @3: B @3
$\rightarrow_B$ @1: G @3 @3, @3: B @3
$\rightarrow_B$ …



@1: G @2 @3, @2: A @3, @3: B @2
$\rightarrow_B$ @1: G @2 @2, @2: A @2
$\rightarrow_A$ @1: G @2 @2, @2: A @2
$\rightarrow_A$ …



In the two resulting graphs there are no other redexes than the self-embedding redexes reducing to themselves, so the two graphs have no common reduct.

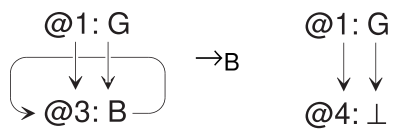
The only case in which this kind of non-confluency occurs is the case where self-embedding redexes do not have a common reduct (Kennaway *et al.*, 1993a). This source of non-confluency can only be taken away by changing the *semantics* of graph rewriting by adding a special **empty node** with the property that a self-embedding redex is reduced to this empty node instead of to itself. Besides removing the source of non-confluency, this solution has the advantage that it allows a categorical description of the semantics with only one push-out instead of two (Kennaway, 1990).

In the operational semantics it means a slight adaptation of step 4 (Section 5.3.3): a special case is inserted for a redirection of a self-embedding redex; in such a case instead of the standard redirection a redirection to an empty node is performed.

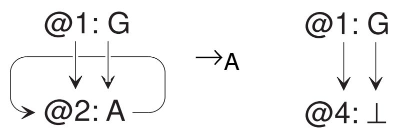
In the sequel it will always be assumed that the semantics is changed in this way. This has as a consequence that self-embedding redexes are not a source of non-confluency any more.

The graph @1: G @2 @3, @2: A @3, @3: B @2 reduced with the new semantics and the same rules described above. Both self-embedding redexes are reduced to an empty node.

@1: G @2 @3, @2: A @3, @3: B @2
$\rightarrow_A$ @1: G @3 @3, @3: B @3
$\rightarrow_B$ @1: G @4 @4, @4: $\perp$



@1: G @2 @3, @2: A @3, @3: B @2
$\rightarrow_B$ @1: G @2 @2, @2: A @2
$\rightarrow_A$ @1: G @4 @4, @4: $\perp$



5.6.2 Sharing and computational complexity

Sharing can be used on the right-hand side of rewrite rules to avoid duplication of work by creating shared subgraphs or even cyclic graphs. Node-ids can be used on the left-hand side to address subgraphs in various ways.

In the following example cyclic objects are reduced to other cyclic objects. It is a solution for a Hamming problem: it computes an ordered list of all numbers of the form 2^n3^m, with n, $m \geq 0$. An efficient solution to this problem can be obtained by means of creating cyclic sharing in the contractum making heavy use of computations already done.

```
Ham  →    x:  Cons 1 (Merge (Map (* 2) x) (Map (* 3) x))
```

The map and merge functions (mapping a function on a list and merging two sorted lists into one sorted list with no multiple elements) can be defined as follows (IF, <, = and * are assumed to be defined elsewhere, Cons and Nil are just constructors with no special (predefined) meaning):

```
Merge   Nil             Nil             →   Nil
Merge   f: (Cons a b)   Nil             →   f
Merge   Nil             s: (Cons c d)   →   s
Merge   f: (Cons a b)   s: (Cons c d)   →   IF    (< a c)
                                                (Cons a (Merge b s))
                                                (IF   (= a c)
                                                      (Merge f d)
                                                      (Cons c (Merge f d)))

Map  f   Nil            →    Nil
Map  f   (Cons a b)     →    Cons (Ap f a) (Map f b)

Ap (* a) b   →    * a b
```

Note the use of Ap for the rewriting of a curried application of * (used as a constructor with one argument) to an application of * as a δ-rule (a function with two arguments). Further note that because f is specified in the redex pattern of Merge, both the components a and b as well as the subgraph f as a whole are available on the right-hand side.

The cyclic solution presented above has a polynomial complexity. The behaviour of this solution is much better than a solution in which recursion was used instead of this cyclic structure. The recursive solution specified below has an exponential complexity because previous computations performed by a call of Ham have to be recomputed.

```
Ham  →    Cons 1 (Merge (Map (* 2) Ham) (Map (* 3) Ham))
```

5.6.3 Copying

GRSs are designed to exploit sharing wherever possible. As GRSs are an extension of TRSs, in GRSs one would expect a possibility of creating copies of subgraphs similar to the creation of duplicates of subterms in TRSs. However, in an ordinary GRS it is not possible to copy an arbitrary unknown data structure since in the contractum pattern multiple references to node-ids always imply sharing. In Chapter 14 an extension of GRSs is discussed that provides copying of arbitrary graph structures.

5.7 Reduction strategies

A **reduction strategy** is a function that, given a GRS and a graph, prescribes which redexes in the graph have to be rewritten next. 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**. A **reducer** is a process that repeatedly 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 redexes any more. A reduction strategy is **normalizing** iff, for any graph, having a normal form, the reducer applying the strategy will terminate at a normal form.

Another property a strategy can have is being *hypernormalizing*. A reduction strategy is **hypernormalizing** iff the strategy is normalizing even if it is diluted with a finite number of arbitrary reduction steps. The notion of hypernormalizing strategies can in the same way be defined in the TRS world and in the λ-calculus. However, the notion is particularly important in the context of *term graph rewriting* (see Section 5.8). It is also an important property when GRSs are used as the basis for a programming language. It makes it possible to deviate from the standard strategy to allow a more efficient evaluation order (see Part 4).

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.

As is the case with TRSs, there exists no normalizing strategy for an arbitrary GRS. Instead of starting another independent quest for subclasses and normalizing strategies for GRSs, it seems to be more natural to investigate GRSs as extensions of TRSs.

5.8 Term graph rewriting

Although it is possible to search for confluent subclasses of GRSs with corresponding normalizing strategies we shall only investigate properties of TRSs that are converted into the GRS world to see whether it is possible to use sharing of subterms in general. Special attention is paid to the conversion of the most important TRS subclass for functional languages: priority rewrite systems with the functional strategy.

Term graph rewriting means that first a TRS and the term to be rewritten is converted (*lifted*) to a GRS; then in the GRS world reduction takes place and a normal form which is a graph is converted again (*unravelled*) to a term in the TRS world (Figure 5.6). Term graph rewriting (Barendregt *et al.*, 1987a) formally connects TRSs with GRSs.

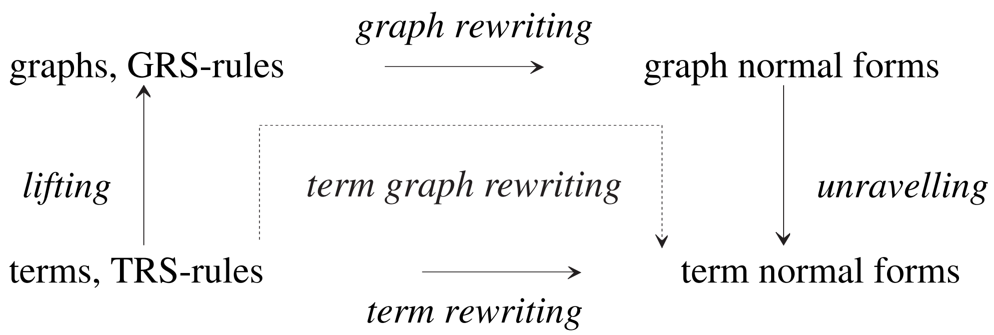


Figure 5.6 Diagram of term graph rewriting.

The conversion of a TRS to a GRS is called the **lifting** of a TRS to a GRS. One can easily convert a TRS to a GRS almost without any syntactic change in the rules, because the TRS notation is also a valid GRS (shorthand) notation. Although the rules of that GRS are syntactically the same as the TRS rules, semantically they get a different meaning. Note that TRSs and GRSs work on different objects and the comparing of terms in TRSs is not at all the same thing as the comparing of node-ids in GRSs. However, many properties of TRSs are inherited if TRSs are lifted. In many cases the only difference between TRSs and GRSs will be that, through sharing, fewer steps are needed to reach a normal form.

The conversion of a graph to a term is called the **unravelling** of the graph. A graph is unravelled by recursively duplicating all shared subgraphs starting from the root of the graph. This yields a tree representation of the term that is the result of the unravelling. When the graph to be unravelled is cyclic, recursive duplication goes on forever resulting in an infinite term.

There are various ways of lifting TRSs and they have led to the investigation of special classes of GRSs. Since the semantics of a comparing rule that is lifted would change too dramatically, these classes have in common that no rule is comparing. This restriction implies that it is impossible to pattern match on equivalency of node-ids (sharing). So a left-hand side in these classes is always a graph without sharing (like a term). Because of this restriction the classes are called **term … rewriting systems** where **…** indicates what kind of right-hand sides are allowed. The most simple subclass is **term tree rewriting systems** (TTRSs) in which right-hand sides must be trees. In order to express this properly a special semantic action for duplication of arguments on the right-hand side is necessary (see Section 5.6.3). TTRSs are equivalent to TRSs.

The most natural subclass of TGRSs is **term dag rewriting systems** (TDRSs) in which right-hand sides must be **dags** (*d*irected *a*cyclic *g*raphs). When a TRS is lifted to a TDRS all rules remain syntactically the same.

The following TRS is syntactically equal to its lifted TDRS:

Double x	→	+ x x
Heavy_Computation	→	Easy_Number

Figure 5.3 shows how the term Double Heavy_Computation is reduced in a TRS (using a lazy strategy). In a TRS four reduction steps are needed. The graph Double Heavy_Computation will be reduced in the TDRS as follows:

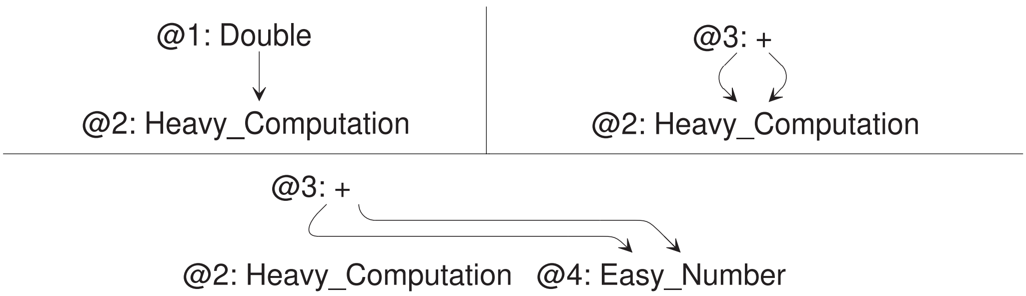


Figure 5.7 Sharing of computations in graph rewriting.

Figure 5.7 shows that in a GRS only three reduction steps are needed to perform the corresponding computation. Due to sharing the Heavy_Computation is only reduced once. The results of both computations, modulo unravelling, are the same.

In Barendregt *et al.* (1987a) it is proven that lifted *orthogonal* TRSs are also confluent as TDRSs. It is shown that if strategies are hypernormalizing for an orthogonal TRS, these strategies are normalizing when they are used in the corresponding TDRS. This can be explained as follows: if a shared subgraph is chosen by a strategy, not only is the GRS counterpart of the original term reduced, but as a side-effect (due to the sharing) also the counterpart of other equivalent terms. If the original strategy is hypernormalizing, the additional rewriting steps that are performed cannot influence the outcome of the computation.

Lifting is not always safe for a *general* TRS. There are some subtle differences between TRSs and corresponding TDRSs. It is possible that the corresponding TDRS has less normal forms than its original TRS.

The normal form properties can change when an *ambiguous* TRS is lifted to a TDRS :

A x	$\rightarrow$	B x x
C	$\rightarrow$	D
D	$\rightarrow$	C
B C D	$\rightarrow$	E

The normal form of A C in the TRS world is E according to A C $\rightarrow_A$ B C C $\rightarrow_C$ B C D $\rightarrow_B$ E (Figure 5.8).

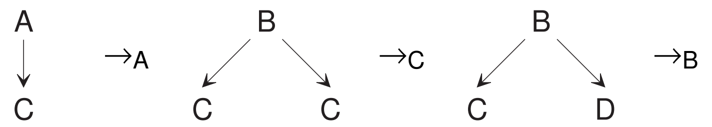


Figure 5.8 Reduction to normal form in the original TRS.

In the graph world the TDRS with the same rules will reduce the graph A C to

@1: A @2, @2: C
$\rightarrow_A$ @3: B @2 @2, @2: C
$\rightarrow_C$ @3: B @4 @4, @4: D
$\rightarrow_D$ @3: B @5 @5, @5: C $\rightarrow_*$ …

There is no normal form. The double occurrence of x on the right-hand side of the A-rule creates sharing that (in this example) will exist forever (Figure 5.9).

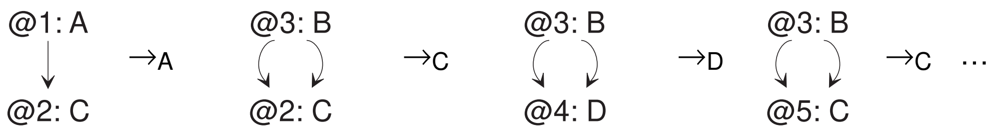


Figure 5.9 Fewer normal forms in the lifted TRS than in the original TRS.

Via term graph rewriting in TDRSs it is proven that, independent of the chosen strategy, sharing terms is *sound* (i.e. an unravelled normal form of the TDRS is also a normal form for the TRS) for all TRSs and *complete* (i.e. for any normal form in the TRS, the TDRS will have a normal form that can be unravelled to it) for orthogonal TRSs.

Well-known strategies that are hypernormalizing for certain sub-classes are, for example, parallel-outermost, leftmost-outermost, the discriminating position strategy and the Huet–Lévy strategy. So a TRS with one of these hypernormalizing strategies can be lifted to a TDRS safely. The result (modulo unravelling) will be the same. In general, due to the sharing of subterms, fewer reduction steps will be needed in the TDRS to reach the normal form.

The most general subclass is the one in which no further restrictions are given: **term graph rewriting systems** (TGRSs). In a TGRS right-hand sides are GRS graphs that may contain cycles. For certain simple recursive rewrite rules in a TRS (see, for example, the solution for the Hamming problem in Section 5.6) rules with cycles on the right-hand side are created in the corresponding TGRS. In this way infinite data structures can be implemented efficiently with cyclic graphs.

Lifted orthogonal TRSs are also confluent as TGRSs (due to the changed semantics introduced in Section 5.6.1). Strategies that are hypernormalizing for orthogonal TRSs are also normalizing when they are used in the corresponding TGRS.

There is one important difference compared with TDRSs. The reduction in the TGRS may yield a normal form that is a cyclic graph, while the corresponding reduction in the TRS does not terminate.

Consider the following TRS:	Lifting this to a TGRS yields:
F x → A x	F x → A x
G → F G	G → x: F x
Start → G	Start → G

The normal form of the TGRS is @N: A @N. In the TRS the computation does not terminate. The term reduces to terms of the following form: A(A(A(...))).

The unravelling of a cyclic graph in normal form results in an infinite term. Note that it is impossible to specify infinite terms in TRSs. However, an infinite term can be seen as a limit of a non-terminating computation. In a TGRS an infinite term can be specified in a cyclic graph. But, if such a cyclic graph normal form is unravelled to a term, the unravelling process does not terminate either. If in an implementation the result of the unravelled graph would be printed it is indistinguishable from the printing of the head normal form parts of the non-terminating computation obtained via term reduction. Both prints produce equal output and do not terminate.

The unravelling of an empty node results in a **bottom** symbol, $\perp$, i.e. a special symbol in the TRS world that represents a non-terminating computation producing no result. Again the results are indistinguishable since in both cases *no* resulting output is produced.

So in that sense also lifting a TRS to a TGRS is *sound* (i.e. an unravelled normal form of the TGRS is also a normal form for the TRS if the unravelled normal form is a finite non-bottom term while the results are indistinguishable otherwise) for all TRSs and *complete* (i.e. for any normal form in the TRS, the TGRS will have a normal form that can be unravelled to it) for orthogonal TRSs (Kennaway *et al.*, 1993b).

Empty nodes in the context of functional languages:

f = h f	F → x: H x
h x = x	H x → x
f?	Start → F

In a functional language a computation corresponding to an empty node is assumed to be not intended. The Miranda program on the left halts with a black hole error. The GRS reduces the initial graph to an empty node. In the functional graph rewriting language Clean (see Chapter 8) a cycle_in_spine error would be given.

So a TRS with a hypernormalizing strategy can also be lifted to a TGRS safely. Due to the presence of cycles even fewer reduction steps may be needed in the TGRS world than in the TDRS world.

5.8.1 Functional graph rewriting systems

FTRSs can also be lifted to a GRS class called **functional graph rewriting systems** (FGRSs): TGRSs with priorities added using the functional strategy. Lifting is accomplished in the same way as explained above. FGRSs will be heavily used in this textbook. In Parts 4 and 5 it is shown that efficient state-of-the-art implementations on sequential and parallel hardware can be obtained by compiling functional languages to Clean, a language that is based on FGRSs.

Adding priorities in GRSs has similar semantics as in TRSs, i.e. a matching rule can only be chosen if no rule with higher priority can ever match on the (internally rewritten) graph. As in TRSs this has the following properties:

- it is undecidable whether a term can ever match a pattern via internal rewrites;
- forcing evaluation leads to a decidable approximation of the priority semantics in which reduction is always correct with respect to the priority semantics; however, in some cases forcing evaluation leads to non-termination, although semantically it was avoidable;
- textual order is defined to correspond to increasing priority;
- the functional strategy effectively applies the discriminating position strategy to one rule alternative at a time.

Factorial as an FGRS (syntactically unchanged by the lifting process):

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

Note that n is shared on the right-hand side of the second alternative.

The definition of the functional strategy has to be slightly adapted for FGRSs. Evaluation is not uniformly forced to strong head normal form but to its analogue in GRSs: strong root normal form. The functional strategy is efficiently implementable, intuitively clear and normalizing for a large class of FGRSs (see the previous chapter). The functional strategy is even hypernormalizing for this class. Of course, only diluted reductions are allowed that are not in conflict with the priority in the rules.

So also lifting a TRS with priorities and the functional strategy to a FGRS is *sound* (i.e. an unravelled normal form of the FGRS is also a normal form for the TRS if the unravelled normal form is a finite non-bottom term while the results are indistinguishable otherwise) for all TRSs and *complete* (i.e. for any normal form in the TRS, the FGRS will have a normal form that can be unravelled to it).

Due to the use of the functional strategy FGRSs can be regarded as a very simple functional language with a more explicit notion of controlling the sharing of terms.

The functional strategy is a partial function for FGRSs

There is a subtle difference between the functional strategy in TRSs and its analogue in FGRSs: owing to the presence of cycles the functional strategy is a partial function for FGRSs. The explanation for this is that it can happen that the functional strategy itself does not terminate when it recursively searches for a redex on a cycle.

Consider the following TRS:	The FGRS resulting from lifting this TRS:
F (F a) → 1	F (F a) → 1
G → F (F G)	G → x: F (F x)
initial term: G	Start → G

In the TRS the functional strategy tries to force the evaluation of the argument of F (after reduction of G). Since the argument of F contains an F again, also this argument is forced. This leads to reduction of G resulting in further applications of F. The reduction process will not terminate.

Also in the FGRS the functional strategy tries to force the evaluation of the argument of F (after reduction of Start and G). Since on the cycle every node consists of F with an argument, the process of selecting a candidate for evaluation will go on for ever. Reduction will never take place.

Note that in both systems termination would have been possible if evaluation was not forced.

Non-termination of the strategy in the FGRS corresponds to non-termination of the computation with the functional strategy in the TRS.

A special aspect of the non-termination of the functional strategy in FGRSs is that non-termination can be detected as an erroneous situation: whenever it occurs, the strategy is running on a cycle.

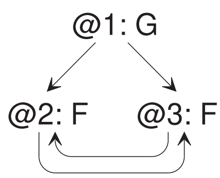
One could be tempted to change the semantics of the functional strategy such that when the strategy runs on a cycle, evaluation is not forced. Instead, a test could be performed to see whether the pattern matches the graph and a redex is indicated accordingly. Then, the strategy becomes a total function. But an important property of the functional strategy would be lost: the strategy would not be hypernormalizing any more. The problem is caused by an interference between partial ambiguity in the rule system and a cyclic graph structure that matches in various ways.

The functional strategy would not be hypernormalizing if it would not always force evaluation:

```
F (F a)  →  1

G 1 x    →  1
G x y    →  G x y
```

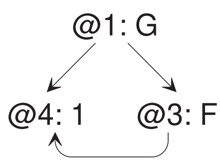
Take the following graph:

```
@1: G @2 @3,
@2: F @3,
@3: F @2
```



Then, the normal form would be 1 since it would be detected that the strategy runs on a cycle: forcing evaluation is stopped, the cycle matches the redex pattern of the F rule and reduction to normal form takes place.

@1: G @2 @3, @2: F @3, @3: F @2
$\rightarrow_F$ @1: G @4 @3, @4: 1, @3: F @4
$\rightarrow_{G_1}$ @4: 1



If such a reduction is diluted with reduction of the second argument of G, then afterwards the functional strategy will not reach the normal form any more:

$$
\begin{array}{ll}
 & \text{@1: G @2 @3, } \underline{\text{@2: F @3}}, \underline{\text{@3: F @2}} \\
\rightarrow_{F} & \underline{\text{@1: G @2 @4}}, \text{ @2: F @4, @4: 1} \\
\rightarrow_{G_2} & \underline{\text{@1: G @2 @4}}, \text{ @2: F @4, @4: 1} \\
\rightarrow_{*} & \dots
\end{array}
$$

$$
\begin{tikzcd}
 & \text{@1: G} \arrow[dl] \arrow[dr] & \\
\text{@2: F} \arrow[rr, bend right] & & \text{@4: 1}
\end{tikzcd}
$$

The functional strategy being a partial function for FGRSs is just a new aspect of the fact that forcing evaluation is a decidable approximation of the priority semantics: for graphs that correspond to infinite computations in the term rewrite world, the non-termination may not only concern the reduction process but also the strategy itself.

5.8.2 GRSs versus TRSs

We have shown that many interesting TRSs can be lifted safely to GRSs. For these systems the outcome of a computation in the TRS world is equal to the unravelled outcome of the corresponding computation in the GRS world. It has been shown that this is not a general property as one might have expected.

Different behaviours between TRSs and GRSs are generally possible when graphs with cyclic structures are being used. Also comparing in TRSs has quite a different meaning than comparing in GRSs and is therefore forbidden when TRSs are lifted to GRSs. But fortunately the lifting process is sound and complete for important (sub)classes of orthogonal TRSs and TRSs with priorities.

As a consequence one can say that the most important difference between these kinds of TRSs and GRSs is that in the latter duplication of work is avoided. This is an efficiency aspect: fewer rewrite steps are needed in the GRS than in the original TRS. Although GRSs can be regarded as an extension of TRSs one should realize that in general it is impossible in a standard GRS to duplicate work explicitly since it is impossible to copy arbitrary graph structures in these systems. An extension of GRSs in which copying as well as sharing can be expressed is given in Chapter 14.

5.9 Generalized graph rewriting systems

A very high expressive power is realized in **generalized graph rewriting systems** (Barendregt *et al.*, 1987b) because the transformations that can be performed are extended. This induces a trade-off: adding restrictions decreases expressiveness but it may yield important properties for reasoning or implementation (e.g. referential transparency or efficient fine-grain parallelism). In a generalized GRS the right-hand side of a rewrite rule may contain several redirections. Any node can be redirected to another one. All redirections specified in the rule are done in parallel in one reduction step.

This extension makes it possible to express much more complex changes of a graph structure than simple subgraph replacement. A generalized GRS can be regarded as a very powerful graph assembler. In such a system confluency is by no means guaranteed. Assignment can be modelled very well. Unification can be expressed. Any concrete machine language can be modelled in such a generalized system. As a consequence, referential transparency is no longer guaranteed.

So generalized graph rewriting can model other languages than pure functional languages, such as logical languages and even imperative languages (Glauert *et al.*, 1987). Although generalized graph rewriting systems are not of direct importance in the context of this textbook they are explained here briefly to illustrate their conceptual power.

5.9.1 Multiple redirections

With multiple redirections side-effects can be expressed. All redirections specified in a rule are done in parallel in one reduction step. As a consequence, parallel redirections act like parallel assignments with which it is possible to interchange values of variables. With parallel redirections it is possible to effectively interchange node-ids.

Illustration of parallel evaluation of multiple redirections:

```
r:  F x y z          →     r := x, y := z
```

Take the following graph `@1: F @2 @2 @1`. This graph matches the rule and the redirections that have to be performed are `@1 := @2, @2 := @1`. Because of the parallel evaluation (like in a parallel assignment) the effect will be that the applied occurrences of `@1` and `@2` are exchanged. So the result will be `@1: F @1 @1 @2`.

Now suppose that with the same rule the graph is `@1: F @2 @1 @3`. Also this graph matches the rule and now the redirections that have to be performed are `@1 := @2, @1 := @3`. Because of the parallel evaluation of the substitutions either all applied occurrences of `@1` will be replaced by `@2` or all applied occurrences of `@1` will be replaced by `@3`. The choice between these two possibilities is made non-deterministically. Note that the semantics does not allow some of the applied occurrences of `@1` to be replaced by `@2` and others to be replaced by `@3`. In this sense a redirection is indivisible.

It is intuitively clear that side-effects can be created with multiple redirections. This will be further illustrated by an example designed by E. Meijer: a GRS that implements a simple unification algorithm, using multiple redirections. It operates on representations of two types, returning ‘cannot unify’ in case of failure. The types are constructed from three basic types `I`(nteger), `B`(oolean) and `V`(ariable) and a composition constructor Com(position). Different type variables are represented by distinct nodes. References to the same shared node are taken to be references to the same type variable. The unification rule has as arguments the two types that have to be unified. During the unification process the graph representing the unification is constructed and delivered as the result. However, it may also turn out that unification is not possible. Then, the graph constructed so far has to be replaced by 'cannot unify' in cases of failure. To make this possible the root of this graph is passed as a third argument such that it can be redirected when unification is not possible.

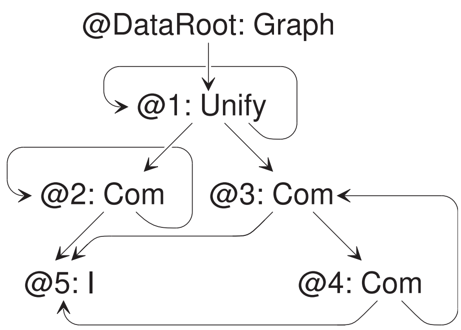
	Unify	x	x	r	→	x
o:	Unify	t1:(Com x y)	t2:(Com p q)	r	→	n: Com (Unify x p r) (Unify y q r), o := n, t1 := n, t2 := n
o:	Unify	t1:V	t2	r	→	o := t2, t1 := t2
o:	Unify	t1	t2:V	r	→	o := t1, t2 := t1
	Unify	(Com x y)	I	r	→	n: "cannot unify", r := n
	Unify	(Com x y)	B	r	→	n: "cannot unify", r := n
	Unify	I	(Com x y)	r	→	n: "cannot unify", r := n
	Unify	B	(Com x y)	r	→	n: "cannot unify", r := n
	Unify	I	B	r	→	n: "cannot unify", r := n
	Unify	B	I	r	→	n: "cannot unify", r := n

For instance, find the unification of a cycle y: Com I y and the same cycle unravelled once z: Com I (Com I z). The result will be u: Com I u. Hence, assume that the start rule is

```
Start →   x: Unify y z x,
          y: Com I y,
          z: Com I (Com I z)
```

After reduction of the start rule there is the following graph:

```
@DataRoot:   Graph @1,
@1:  Unify @2 @3 @1,
@2:  Com @5 @2,
@3:  Com @5 @4,
@4:  Com @5 @3,
@5:  I
```



It can be rewritten as follows by applying the second rule for Unify:

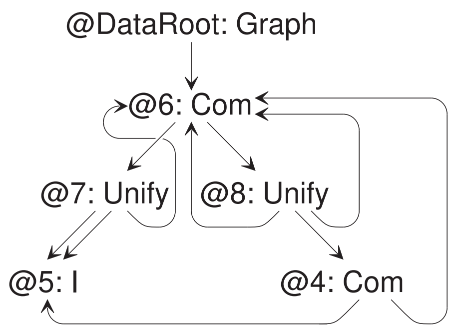
μ(o) = @1, μ(t1) = @2, μ(t2) = @3, μ(x) = @5, μ(y) = @2, μ(p) = @5, μ(q) = @4, μ(r) = @1

The contractum is:

@6: Com @7 @8, @7: Unify @5 @5 @1, @8: Unify @2 @4 @1

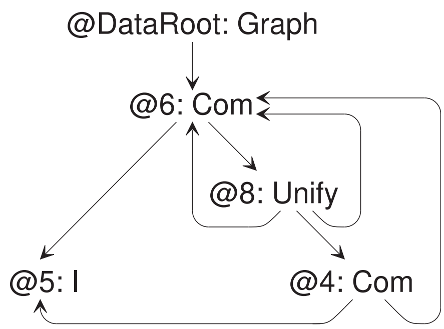
After redirection and garbage collection this yields

@DataRoot: Graph @6,
@6: Com @7 @8,
@7: Unify @5 @5 @6,
@5: I,
@8: Unify @6 @4 @6,
@4: Com @5 @6



Rewriting this graph by applying the first rule for Unify to @7 yields:

@DataRoot: Graph @6,
@6: Com @5 @8,
@5: I,
@8: Unify @6 @4 @6,
@4: Com @5 @6



This can be rewritten by applying the second rule for Unify to @8:

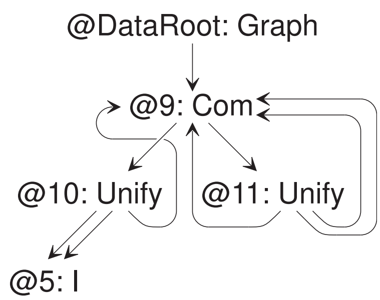
μ(o) = @8, μ(t1) = @6, μ(t2) = @4, μ(x) = @5, μ(y) = @8, μ(p) = @5, μ(q) = @6, μ(r) = @6

The contractum is

@9: Com @10 @11, @10: Unify @5 @5 @6, @11: Unify @8 @6 @6

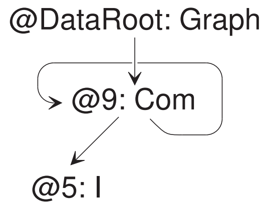
After redirection and garbage collection this yields

@DataRoot: Graph @9,
@9: Com @10 @11,
@10: Unify @5 @5 @9,
@5: I,
@11: Unify @9 @9 @9



Finally, on this graph the first rule for Unify on @10 and on @11 can be applied, yielding the wanted unifier:

```
@DataRoot:  Graph @9,
@9:  Com @5 @9,
@5:  I
```



Of course, this small example does not model unification in general. But, it gives an idea of the power of redirection and how it might be used to solve these kinds of problems.

5.10 GRSs as a basis for functional languages

In the previous chapter we have seen that there are several kinds of TRSs that can be used as a basis for functional languages. Most properties of a TRS are inherited if the TRS is lifted to a GRS. The most important difference between TRSs and GRSs is that in the latter duplication of work is avoided. This makes GRSs more suited to serve as a basis for functional languages if efficiency is of interest.

As with TRSs, if a functional program is translated to a GRS the complexity of the compilation scheme, the complexity of the obtained code and the efficiency of the reducer are heavily influenced by the kind of GRS chosen. For a lazy functional language without explicit graph rewriting semantics one has to assume that sharing is created and that for certain recursive definitions a cyclic structure is created (which in fact means that assumptions are made about some *unknown* graph rewriting semantics). These assumptions may be essential not only for reasoning about the efficiency of the program but in certain cases even for reasoning about the computational complexity of the program (see Section 5.6.2).

The most obvious choice is to use FGRSs as a basis. Owing to the use of the functional strategy FGRSs can be regarded as a very simple functional language with a more explicit notion of controlling the sharing of terms. Part 4 treats the translation from Miranda to the FGRS based language Clean.

5.11 GRSs as a basis for implementations

In practice almost all implementations of functional languages use sharing in one way or another. Many implementations (e.g. of HOPE, Haskell, LML and Miranda) can therefore be regarded as implementations of GRSs. For instance, the implementation of Sasl, described in Turner (1979a), is based on the following combinatorial GRS:

```
Ap (Ap (Ap S x) y) z   →   Ap (Ap x z) (Ap y z)
Ap (Ap K x) y          →   x
Ap I x                 →   x
```

Note that in the S-rule sharing of the distributed argument z is maintained. Leftmost-outermost is hypernormalizing for a combinatorial TRS and therefore this strategy can be used as normalizing strategy for a combinatorial GRS also. Of course, when an implementation is made for a GRS, special attention is paid to make such an implementation as fast as possible. This may lead to exploiting stacks, registers, δ-rules, avoiding building of graphs whenever possible, implementation of redirection via overwriting of nodes or via creation of indirection nodes, etc. In Parts 4 and 5 of this book we show in detail how efficient state-of-the-art implementations of the FGRS-based language Clean is achieved.

Summary

- In GRSs *graphs* are being rewritten instead of *terms*. This makes it possible to *share* computations during rewriting such that duplication of work can be avoided.
- There are many similarities between TRSs and GRSs; many notions like *normal forms*, *strategies* and *rewriting* in GRSs are similar to these notions in TRSs; many properties of GRSs are similar to properties of TRSs:
  - *normal forms* are not *unique*;
  - the *root normal form property* is *undecidable*;
  - the *strong root normal form property* is *decidable*;
  - there is *no normalizing strategy*.
- There are some important differences between TRSs and GRSs:
  - in GRSs connected, possibly cyclic *graphs* are being rewritten instead of *trees* (*terms*);
  - *comparing* in GRSs implies a test that checks whether two subgraphs are shared or not; two syntactically equivalent subgraphs are considered to be unequal in GRSs if they are not shared; in TRSs two syntactically equivalent subterms are always considered to be equal.
- Ambiguity is a source of non-confluency in GRSs. *Comparing* is conjectured to be not a source of non-confluency. The introduction of *empty nodes* removes *self-embedding redexes* as a source of non-confluency.

- A TRS can be *lifted* to a GRS; this transformation can be performed almost without any change in syntax.
  - Owing to the difference in semantics comparing is not allowed in TRSs that are lifted.
  - If lifting is sound and complete, reduction can be performed in the GRS world instead of the TRS world; if the resulting graph is converted back to a term the result is equivalent with the result that would have been obtained in the TRS world; owing to the sharing the number of reduction steps that is needed to reach the normal form is generally decreased.
- A TRS can be lifted to a GRS without cycles (resulting in a *term dag rewriting system*) or to a GRS with cycles (resulting in a *term graph rewriting system*).
  - Strategies that are *hypernormalizing* for an orthogonal TRS are *normalizing* when they are used in the corresponding TDRS/TGRS: parallel-outermost, leftmost-outermost, the discriminating position strategy and the Huet–Lévy strategy are hypernormalizing and hence normalizing in a lifted context.
  - When a TRS is lifted to a TDRS/TGRS, sharing of terms is *sound* for all TRSs and *complete* for orthogonal TRSs; these properties are independent of the reduction strategy.
  - Lifting a FTRS to a FGRS is *sound* and *complete*.
- Graph rewriting can be *generalized* in such a way that *non-functional concepts* as found in logic languages and imperative languages can also be modelled.
- Graph rewriting semantics is required to make it possible to *reason* about the actual *computational complexity* of a lazy functional program.
- The class of FGRSs seems best suited to be used as a computational model for functional languages; important concepts in FGRSs also play an important role in functional languages:
  - *pattern matching* is available;
  - *sharing* is used as an optimization in the implementation;
  - reduction proceeds with the *functional strategy*.

EXERCISES

5.1 Write the GRS-rules in their canonical form.

(a) F (F 5) → G 5 5
(b) F x y → G x (F x y)
(c) F x:5 x → G x (H 5)

5.2 Write the GRS-rules in shorthand form, as short as possible.

(a)
```
r: F x,
x: Cons x y,
y: F r          →    a: G r b,
                     b: F a
```
(b)
```
r: F x x,
x: Cons x r     →    a: Cons b c,
                     b: Cons c b,
                     c: Cons b r
```
(c)
```
r: F x,
x: Cons y z,
z: Nil          →    y
```

5.3 The following GRS and initial graph is given:

```
Eq x x    → True
Eq x y    → False
F (G x y) → G (F x y) y
```
initial graph: @0:F @1:(G @2:(Eq @3:5 @4:5) @1)

Show all redexes in the graph. Give, for each redex, the mapping from the corresponding rule to the graph. Rewrite, step by step, the redex corresponding to the rule for F (the third rule).

5.4* Write a GRS-rule generating a list of pairs, each pair containing a natural number (in ascending order) and a reference to the pair in the list containing half of that number (rounded downwards). So

Cons a:(Pair 0 a) (Cons b:(Pair 1 a) (Cons c:(Pair 2 b) (Cons d:(Pair 3 b) ...

is the initial segment of this list.

5.5 Write a GRS-rule that traverses a tree from left to right, yielding a list of all nodes in this tree in the order in which they are visited. The tree is built with the constructor Tree, taking two arguments, and the constructor Leaf, which has no arguments (e.g. Tree Leaf (Tree Leaf (Tree Leaf Leaf)) is a tree).

5.6 Give the possible results of the following systems both when you consider them as TRSs and when you consider them as GRSs.

(a)
```
F x          →    G x x
G (+ 0 1) 1  →    0
```
initial expression: F (+ 0 1)

(b)
```
F g (Cons a b)  →  Cons (g a) (F g b)
F g Nil         →  Nil
```
initial expression: F (+ 1) (Cons 1 (Cons 0 Nil))