Chapter 14
Parallel graph rewriting

This chapter will show that processes and inter-process communication can be modelled by extending the GRS model presented in Chapter 5. For reasons of simplicity this extension will be restricted to TGRSs. The extensions made to the TGRS model are inspired both by the concurrency primitives that we would like to have on the language level as well as by the target architectures we are aiming for (see Chapter 13).

Owing to the absence of global memory on loosely coupled parallel architectures it is no longer realistic to share graphs across processor boundaries. In a concrete implementation graphs have to be copied from one local processor memory to another. This copying is so important and complex that we do not want to regard it as a minor implementation issue. So we want to obtain this behaviour as a basic aspect in the underlying model of computation. In order to be able to copy graphs in a flexible way TGRSs are extended with the concept of *lazy copying*.

Furthermore, we want to be able to specify *dynamic process creation* explicitly in a TGRS. For this purpose the strict annotations as introduced in Clean (Chapter 8) are generalized to *strategy annotations* that influence a single incarnation of a reduction strategy and *process annotations* that create new incarnations of a reduction strategy. However, it has to be sound to consider a graph rewrite step as an indivisible step in order to make reasoning about rewriting systems possible. This implies that with process annotations only *interleaved* processes can be created. Still, of course, one would like to be able to create reducers of which the rewrite steps can actually be performed in *parallel* instead of *interleaved*. A special combination of process annotations and lazy copying does allow such a parallel evaluation.

Section 14.1 introduces *strategy annotations* and *process annotations* to control the sequential and the concurrent evaluation order. With process annotations *interleaved*, executing reducers can be created. In

Section 14.2 *copying* and *lazy copying* are introduced. Section 14.3 shows that lazy copying and process annotations can be seen as basic primitives with which higher level annotations such as {Par} and {Self} (as introduced in Chapter 13) can be created. Similar annotations, {P} and {I}, are introduced to create processes that reduce to *root normal form*. These annotations are more primitive, but they allow finer control over the distribution of data and work (see also Chapter 15).

14.1 Annotations to control the reduction order

This section introduces *strategy annotations* and *process annotations* in TGRSs as generalizations of the strict annotations used in Section 8.3.

The semantics of strategy and process annotations on a *right-hand side* are effected after a rewrite of the corresponding rule alternative but before the function at the root of the right-hand side is considered for evaluation. In the case that in a single right-hand side more than one annotation is used, the order in which these annotations are dealt with takes the graph structure into account. The annotations are in principle effected depth-first. In order to obtain a higher degree of flexibility the annotations in this section are parametrized with a strategy σ.

14.1.1 Strategy annotations in TGRSs

In TGRSs strategy annotations, denoted by $\{!_\sigma\}$, can be placed at arguments on the right-hand side of a rewrite rule.

The semantics of a **strategy annotation in a TGRS** is as follows. When the current strategy considers as a candidate for reduction a function application that has an argument which is annotated in the rewrite rule with a strategy annotation σ, then the current strategy is *temporarily* interrupted: instead the strategy σ is applied on the subgraph corresponding to the argument. The interrupted strategy will continue again (e.g. with another annotated argument or with the application of the function itself) when no redexes are indicated by σ, i.e. when the subgraph is in **σ normal form**.

Looking at the overall strategy function one can observe that it is composed of several strategy functions that invoke each other. Each of them is invoked on the indicated subgraph where an annotation occurs in the contractum pattern. However, at each moment only *one* strategy σ is actively applied; all the others are interrupted.

We distinguish three kinds of *strategy annotation*:

- The **rnf annotation** {!} invokes the *functional strategy to strong root normal form* (used, for example, in Section 8.3).
- The **nf annotation** {!!} invokes the *functional strategy to normal form* (used for the evaluation of the Start node (see Section 11.2)).

- The **step annotation {*}** is used to invoke a *zero or one step reduction* strategy function that performs one rewrite on the annotated function application if it is a redex (used, for example, to force copying of graphs in Sections 14.3 and 14.4).

14.1.2 Graph rewriting with interleaved semantics

When a *parallel* reduction strategy is used, *several* redexes are indicated as candidates for reduction (see Section 5.7). However, in principle it is not allowed to rewrite all these redexes simultaneously. Unpredictable graphs may be constructed if matching, contractum construction, redirection and garbage collection of several rewrites were to take place in parallel. Therefore, without additional restrictions it is *a priori not* sound to rewrite in *parallel*, even when a so-called *parallel reduction strategy* is being used. In order to be able to reason about the result of reductions indicated by a parallel strategy, it is assumed that 'parallel' rewriting has interleaved semantics, i.e. parallelism is *simulated* by *interleaving* ordinary, *sequential* rewrite steps. The rewrite step itself is considered to be *indivisible*. So when a parallel reduction strategy is used, only one of the indicated redexes is chosen in an arbitrary way and rewritten as usual. Then the parallel reduction strategy will determine the next candidates. It will always be assumed that the parallel strategy will at least offer the not-chosen candidates for the next round. To obtain a proper simulation of parallel reduction it is assumed that the rewriting will be **fair**, i.e. a redex repeatedly indicated by the parallel strategy will *eventually* be rewritten by the reducer.

Section 14.3 will discuss cases for which the *interleaved* semantics is not required, allowing a more realistic *truly parallel* evaluation.

14.1.3 Process annotations in TGRSs

By using **process annotations** in the rewrite rules one can indicate on which part of the graph strategy σ has to be applied in an *interleaved* way. As with strategy annotations, several sequential strategy functions are invoked at several places (subgraphs) in the graph. The difference with strategy annotations is that for a process annotation a *new* incarnation of a sequential reduction strategy is created such that *multiple strategy incarnations* now *remain* active. These strategies together form a **parallel strategy**: each of them delivers one candidate for rewriting. The parallel strategy thus constructed is completely determined by the sequential strategies that are activated and by the set of root nodes of the several subgraphs on which they are applied.

A reducer is a process that repeatedly chooses and rewrites one of the redexes indicated by the (parallel) reduction strategy. In the case of a parallel strategy specified by using process annotations one can equally well assume that there are several **interleaved reducers**, where each reducer is attached to one of the sequential strategies that the parallel strategy is composed of. So TGRSs are extended in the following way. For each sequential strategy σ on a right-hand side of a rewrite rule process annotations {p_σ} can be specified.

Example of {p_σ} annotation on a right-hand side:

```
Fib 0   →   1
Fib 1   →   1
Fib n   →   + ({pσ} Fib (– n 1)) ({pσ} Fib (– n 2))
```

A process annotation changes the default sequential strategy into a parallel one by applying the sequential strategy σ on the indicated subgraph as well. As a result a new sequential reducer is created with the following properties:

- the new reducer reduces the corresponding subgraph with strategy σ and dies when no more redexes are indicated;
- the new reducer proceeds *interleaved* with the parent reducer;
- all rewrites are *indivisible* actions.

Locking of reducers

A process created with a process annotation may hit on a shared redex which is already under reduction by another process. We have already argued in Chapter 13 that in such a situation it would be best for efficiency reasons to wait until the first process is finished, i.e. has delivered its σ normal form. Therefore we assume that a process that needs information which is already being calculated by another process is **suspended** (**locked**) until the result becomes available.

Such a locking has to be realized by the overall strategy function. Each of the incarnations of a sequential strategy delivers a candidate for rewriting. But when the same redex is indicated by more than one incarnation, only one of them is allowed to reduce the redex until the intended σ normal form is reached. The others are not chosen and locked in that way. *From now on we only consider process annotations that invoke such locking strategies.*

In this chapter two special (locking) *process* annotations are used to model higher level process behaviour like, for example, the behaviour induced by the {Par} and {Self} annotations of Chapter 13.

- The **p-rnf process annotation {$p_!$}** invokes a *process* with the *functional strategy* to *strong root normal form*.
- The **p-nf process annotation {$p_{!!}$}** invokes a *process* with the *functional strategy* to *normal form*.

14.1.4 Deadlock

When processes need information from each other in order to continue a **deadlock** situation may be created (i.e. none of the processes involved can make any progress). This is easily avoided by the programmer through the following important property. When all process annotations occur on positions that are needed for the sequential evaluation, the following holds: if, due to the process annotations, a deadlock situation occurs, ignoring the process annotations would have led to a sequential cycle-in-spine error.

Correspondence between deadlocks of processes and cycle-in-spine errors:

```
Start   →   x: {p!} F y,
            y: {p!} G x

F 0     →   1
G 0     →   1
```

The processes created will be in deadlock because they need each other's result. When the annotations are ignored a cycle-in-spine error occurs.

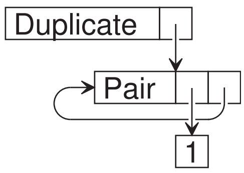
14.2 GRSs with lazy copying

TGRSs have the advantage over TRSs that by sharing it is possible to avoid duplication of work. But in order to model the inter-processor communication on a distributed machine architecture a facility to express graph *copying* has to be added to TGRSs. One might expect that it is already possible to express copying in (T)GRSs. However, this is not true: only a very limited kind of graph copying can be expressed.

Clearly, the following function does not copy its argument since it creates two pointers to the argument instead of duplicating the argument itself.

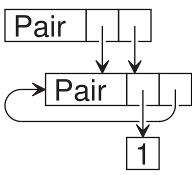
```
Duplicate x   →   Pair x x
```

```
@1:  Duplicate @2,
@2:  Pair @3 @2,
@3:  1
```



This graph reduces to

```
@4:  Pair @2 @2,
@2:  Pair @3 @2,
@3:  1
```



In order to copy a node in a GRS one has to define a rewrite rule in which the pattern is used to determine the node symbol such that a new copy of this node containing the same symbol can be created on the right-hand side. Such a copying rewrite rule is needed for each kind of node that has to be copied. Since it is not possible to match on an arbitrary symbol one cannot define a rewrite rule that can copy an *arbitrary* unknown graph structure. In a special case, i.e. when all the possible symbols of the nodes to copy are known, rules can be defined that can copy a graph structure composed of these nodes. But even in that case, copying remains difficult when shared graph structures are involved. The different sharing structures have to be distinguished. This is only possible in comparing systems. For this purpose the theory of comparing systems would have to be further developed (see Section 5.6.1).

So for reasons of expressiveness and of efficiency of communication it is necessary somehow to include the possibility of copying arbitrary graph structures in the standard semantics of GRSs. This requires in some way finding a construct with semantics in between the semantics of TRSs (in which multiple occurrences of variables on the right-hand side always indicate that duplicates are made) and the semantics of GRSs (in which multiple occurrences of variables on the right-hand side always indicate that a structure with sharing has to be made).

14.2.1 Eager copying

Such a construct is created by extending graph rewriting systems with a function Copy which produces as its result a *full* copy of its argument graph. This function is defined by extending the semantics of graph rewriting. The moment at which this function is executed is, of course, indicated by the reduction strategy. A node containing an application of the Copy function is also called a **copy node**.

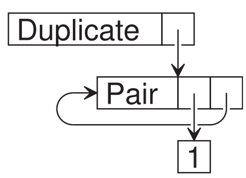
The way of copying defined in this section is called **eager copying**, in contrast to *lazy copying*, which will be defined in the next section.

With the following Duplicate rule an arbitrary graph structure can be copied:

```
Duplicate x    →    Pair x (Copy x)
```

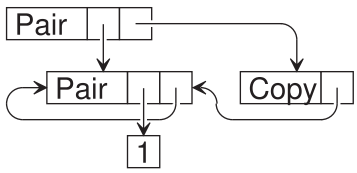
The cyclic graph @2: Pair @3 @2, @3: 1 can be copied as follows:

```
@1:    Duplicate @2,
@2:    Pair @3 @2,
@3:    1
```



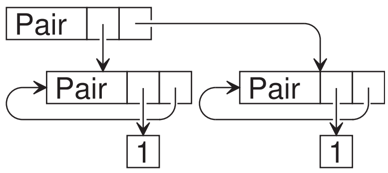
which reduces first to:

```
@4:     Pair @2 @5,
@2:     Pair @3 @2,
@3:     1,
@5:     Copy @2
```



and then to:

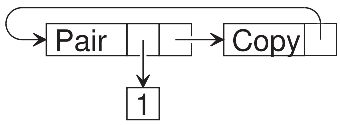
```
@4:     Pair @2 @12,
@3:     1,
@2:     Pair @3 @2,
@12:    Pair @13 @12,
@13:    1
```



Copy nodes are not treated specially during graph rewriting. Consequently, when the argument of a copy node is a cyclic graph containing again the very same copy node, this node is copied with the rest of the cyclic graph. The copy node will occur again in the result of the copy.

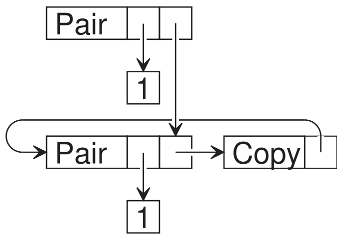
Take the following cyclic graph:

```
@1:     Pair @2 @3,
@2:     1,
@3:     Copy @1
```



The result after executing the Copy function:

```
@1:     Pair @2 @11,
@2:     1,
@11:    Pair @12 @13,
@12:    1,
@13:    Copy @11
```



Semantics of eager copying

The Copy function uses the internal structure of its argument to create the result. The copy it makes is a full isomorphic copy of its argument graph. We saw that possible occurrences of the Copy function itself do not play a special role: they will also be copied. An application of the Copy function is treated just as any other function. Also, with respect to the evaluation order no special treatment is necessary. Evaluating an application of the Copy function simply yields a copy of its argument with which evaluation can continue.

Using the terminology of Chapter 5 the operational semantics of eager copying is defined as follows as an extension of the standard semantics of graph rewriting defined in that chapter. Suppose there is a graph γ over a set of symbols F and a set of node-ids N with contents function C_γ and a set of rules ρ. F is extended with a new symbol: Copy, the set of rules ρ is extended with a rule with left-hand side r: Copy x.

Then, rewriting a graph in the case of the redex being an application of the Copy function takes place according to the following steps (see also Sections 5.3.3 and 5.6.1):

(1) Suppose an application of the Copy function is chosen as a redex with matching mapping μ, i.e. $C_\gamma(\mu(\mathrm{r})) = \mathrm{Copy}\ \mu(\mathrm{x})$

(2) Then, the contractum is constructed as follows: using new node-ids, i.e. node-ids not present in γ, a graph γ' is constructed which is equivalent to the subgraph of $\mu(\mathrm{x})$ in γ.

(Owing to the equivalence and the use of new node-ids γ' is in fact the required copy of the subgraph of $\mu(\mathrm{x})$ in γ.)

(3), (4) and (5) In the standard way, the graph γ' is united with γ, $\mu(\mathrm{r})$, redirected to the root of γ' and garbage collection takes place.

So the function Copy produces as its result a graph copy of its argument.

14.2.2 Lazy copying

Copying *redexes* corresponds to copying *work*. Copying *normal forms* (or certain parts of the graph that will not be reduced anymore) corresponds to copying *data* (in many cases the result of work that has been performed). However, using eager copying the choice is either to share a graph to avoid duplication of work or to copy such a graph which will include a copy of *all* its redexes. In this section we aim to do something in between: *copying of graphs without duplicating work*. Duplication of work can be avoided by maintaining the sharing with the original graph as long as the subgraph to copy has not been evaluated. So the intuitive idea is to *defer* the copying action as long as the work is not finished. After the work is done the copy can be made.

Copying with the choice of breaking up the sharing or maintaining the sharing by deferring the copying is called **lazy copying** (Van Eekelen,1988; Van Eekelen *et al.*,1991). For this reason a node in the graph is extended with the ability to have a **defer attribute**. A node which has this defer attribute is accordingly called a **deferred node**.

The defer attribute is generally assigned to and removed from a node by the reduction strategy. A natural way to do this is to create nodes with the deferred attribute, if this is indicated, on a right-hand side of a rule. When such a deferred node is rewritten it seems reasonable to make the result of the rewrite inherit the deferred attribute, since the 'work' is still going on. Furthermore, it seems sensible to remove the deferred attribute when a node is in σ normal form, since then the 'work' is done. *From now on we shall always assume that every reduction strategy deals with deferred nodes in this way.*

For lazy copying, in the same way as in the previous section, a new primitive can be defined: a function LCopy, which produces as its result a partial copy of its argument graph in which the copying is deferred for those nodes that have the defer attribute. A node containing an application of the LCopy function is also called a **lazy copy node**.

So when the function LCopy hits a deferred node the **copying is deferred**, i.e. stopped *temporarily* until this node loses its defer attribute. This is done as follows: instead of making a copy of the deferred node, a node is created with an application of the function LCopy with the deferred node as its argument. The semantics of the LCopy function deals with deferring the copy in that case simply by the fact that an application of the LCopy function with, as its argument, a deferred node is not considered to be a redex. After creating such a node, the original LCopy function continues as if the deferred node (with its descendants) was copied.

Semantics of lazy copying

Like the Copy function, the LCopy function uses the internal structure of its argument to create the result. Again, possible occurrences of the LCopy function itself do not play a special role: they will also be copied.

Using the terminology of Chapter 5 the operational semantics of lazy copying is defined as follows as an extension of the standard semantics of graph rewriting defined in that chapter. Suppose there is a graph γ over a set of symbols F and a set of node-ids N with contents function C_γ and a set of rules ρ. F is extended with a new symbol: LCopy, the set of rules ρ is extended with a rule with left-hand side r: LCopy x. Furthermore, a predicate P will be defined as a function from N to $\{True, False\}$ indicating for each node whether the node has the deferred attribute or not. The actual concrete definition of P is determined by the reduction strategy. For this purpose, syntactically on a right-hand side nodes can be attributed with a subscript d indicating to the reduction strategy that the corresponding created node has to be deferred. The **non-deferred 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 non-deferred nodes in g that are reachable from n via non-deferred nodes only.

Then, rewriting a graph in the case of the redex being an application of the LCopy function takes place according to the following steps (see also the previous section and Sections 5.3.3 and 5.6.1):

(1) An application of the LCopy function can only be chosen as a redex *if the corresponding argument does not have the deferred attribute*.

Now suppose this is the case: there is a matching mapping μ, i.e. $C_\gamma(\mu(\text{r}))$ = LCopy $\mu(\text{x})$ and $\mu(\text{x})$ does not have the deferred attribute.

(2) Then the contractum is constructed as follows: using for the node definitions new node-ids, i.e. node-ids not present in γ, a graph γ' is constructed which is equivalent to the non-deferred subgraph of $\mu(\text{x})$ in γ.

Since γ' is equivalent to the non-deferred subgraph of $\mu(\text{x})$ in γ and new node-ids are used for the node definitions only, γ' can contain nodes n that have arguments that are deferred nodes in γ. Let δ be the (not connected) graph containing the node definitions of all such nodes n of γ' (δ contains precisely those nodes that have deferred arguments for which copying has to be deferred).

Now, for all nodes n in δ, each deferred argument x of n is redirected in γ' to a node with a new node-id y and the following definition: y: LCopy x. In this case new node-ids are node-ids not present in γ nor in γ' (it is easy to check that the order in which this is done does not influence the result).

This results in a new graph γ''.

(3), (4) and (5) In the standard way, the graph γ'' is united with γ, $\mu(\text{r})$ is redirected to the root of γ'' and garbage collection takes place.

So the function LCopy produces as its result a (possibly partially deferred) graph copy of its argument.

A lazy copying example reduced with the functional strategy.

Start	$\rightarrow$	Duplicate (Fac$_\text{d}$ 6)
Duplicate x	$\rightarrow$	Pair (LCopy x) x
Fac 0	$\rightarrow$	1
Fac n	$\rightarrow$	* n (Fac (– – n))

The following rewrites occur (see also Figure 14.1):

@1: Start			
$\rightarrow$	@2: Duplicate @3,	@3: Fac$_\text{d}$ 6	
$\rightarrow$	@4: Pair @3 @5,	@5: LCopy @3,	@3: Fac$_\text{d}$ 6
$\rightarrow^*$	@4: Pair @6 @5,	@5: LCopy @6,	@6: 720$_\text{d}$
=	@4: Pair @6 @5,	@5: LCopy @6,	@6: 720
$\rightarrow$	@4: Pair @6 @16,	@6: 720,	@16: 720

Note that the d attribute was inherited when the node @3 was redirected to @6 which corresponded with the reduction of the node. The deferred attribute of the node @6 is removed when the root normal form is reached.

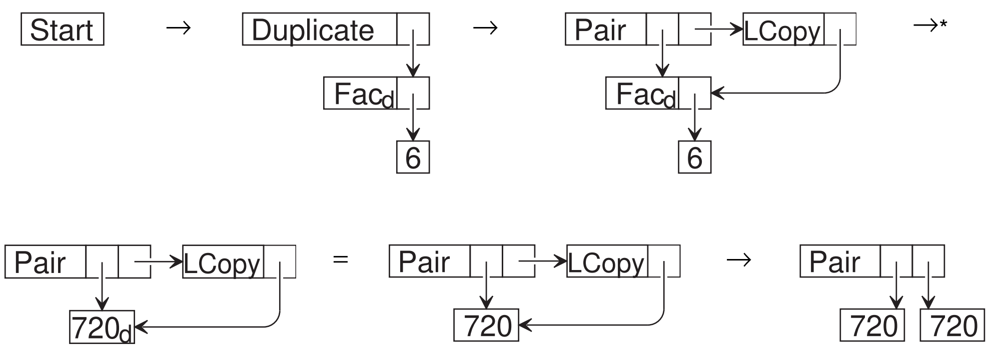


Figure 14.1 Example of lazy copying.

14.2.3 Properties of TGRSs extended with lazy copying

The semantics of graph rewriting with lazy copying is in some sense in between the semantics of term rewriting and the semantics of graph rewriting. The choice can be made either to *share* or to *copy*. When no copying is used a GRS is obtained, when all arguments are copied a TRS is obtained. But the choice can be made in a very flexible way since the copying can be demanded and deferred at any node.

An interesting aspect of lazy copying is that normal forms do not contain defer attributes nor LCopy nodes. In a normal form every subgraph is trivially in σ normal form. Evaluation of nodes to σ normal form eliminates the defer attributes such that all strict applications of the LCopy rule can be reduced.

The following rules:	result in the following normal forms:
Start $\rightarrow$ x: Pair 1 x	@1: Pair 1 @1
Start $\rightarrow$ x: Pair$_d$ 1 x	@1: Pair 1 @1

Evidently, (lazy) copying influences the normal forms in the graph world in a trivial way, since copies are made which mean that the resulting graphs are not graph equivalent anymore. However, the normal forms in the graph world can also be influenced in a more subtle way, since sharing can be broken up when a cycle is copied which contains LCopy nodes or deferred nodes. The result will be partly unravelled with respect to a full copy (a similar property holds for eager copying).

Partial unravelling (and in both cases an infinite normal form) due to copying:

```
Start → x:  Pair 1 (LCopy x)    @1:  Pair 1 @11,   || a cycle with a copy in it
                                @11: Pair 1 @21,   || yields a partly unrav-
                                @21: LCopy @11     || elled cyclic structure.
```

```
Start → x: Pair_d 1 (LCopy x)    @1:  Pair 1 @11,   || when the node is not
                                 @11: Pair 1 @21,   || deferred anymore the
                                 @21: LCopy @11     || result will be yielded.
```

Furthermore, the obtained normal form in the graph world is influenced by the order of evaluation (and hence by annotations). If deferred nodes are not reduced before an attempt to copy them is made, the result will be partly unravelled.

Normal forms of TGRSs with lazy copying are influenced by the reduction order. Consider the following rules:

```
Start →   r: LCopy (A x z),
          x: I_d z,
          z: B
I x   →   x
```

After the Start rule has been applied there are two possible reduction orders that can be taken by a reduction strategy: First one can do the lazy copy.

```
@1:    LCopy @2,
@2:    A @3 @4,
@3:    I_d @4,
@4:    B
```

The identity function cannot be copied because it is deferred. This is remembered by the new LCopy node. Now the identity rule is applied. Then, the deferred property can be removed since the result is in σ normal form. Finally the deferred copying can be completed.

```
@12:   A @13 @14,          @12:   A @13 @14,
@13:   LCopy @3,           @13:   LCopy @4,
@14:   B,                  @14:   B,
@3:    I_d @4,             @4:    B_d
@4:    B

@12:   A @13 @14,          @12:   A @24 @14,
@13:   LCopy @4,           @24:   B,
@14:   B,                  @4:    B
@4:    B
```

The alternative is to do first the identity rule and then the lazy copy. This leads to another graph normal form (shown below). So depending on the reduction order chosen by the strategy, we have two different graph normal forms. The unravellings of the two normal forms are equivalent.

```
@1:  LCopy @2,
@2:  A @3 @4,
@3:  Id @4,
@4:  B
```

```
@1:  LCopy @2,
@2:  A @4 @4,
@4:  Bd
```

```
@1:  LCopy @2,
@2:  A @4 @4,
@4:  B
```

```
@12:    A @14 @14,
@14:    B
```

The unravelling of the normal forms of a rule system with lazy copying will *always* be the same as the unravelling of the normal forms of the same rule system without lazy copying (Barendsen and Smetsers, 1992; Smetsers, 1993). In other words, unravelling is *invariant* with respect to lazy copying. This is a very important property for term graph rewriting and for the implementation of functional languages.

An important property for implementations based on lazy copying has to do with copying LCopy nodes. It can happen that the evaluation of the LCopy itself hits on an LCopy node that refers to a deferred node. Then according to the semantics for that deferred node a new LCopy node is created as usual. This, however, may lead to chains of LCopy nodes. An optimized implementation can do the following: if the LCopy action hits an LCopy node referring to a deferred node, only one new LCopy node is created referring directly to the deferred node. It can be proven that this leads to semantically equivalent results.

14.3 Modelling parallel graph rewriting

This section considers FGRSs only. Parallel instead of interleaved reduction is possible when it is guaranteed that the parallel rewrite steps cannot interfere with each other. It will be shown that one can describe the divide-and-conquer type of parallel graph reduction on distributed architectures by using a special combination of lazy graph copying and p-nf process annotations. This idea is extended to a more general method that can be used to model the {Par} and {Self} annotations introduced in the previous chapter. A variant of this method using p-rnf process annotations is shown to have even greater flexibility for modelling all kinds of process behaviour. Due to the use of lazy copying one can control when graphs are copied from one processor to another while it is possible to ship either work (redexes) or data (root normal forms).

14.3.1 Divide-and-conquer parallelism

With the help of strategy annotations, process annotations and lazy copying we are going to model divide-and-conquer parallelism suited for parallel evaluation on a distributed architecture. For divide-and-conquer evaluation, work (a redex) has to be copied to another processor.

When the work is finished the result (a normal form) has to be shipped back to the original processor. So lazy copying has to be performed *twice*: one lazy copy is directly made when a parallel process is created and one lazy copy is made later when the result is demanded by the parent process. The redex gets the defer attribute to ensure that it is not copied back before the normal form is reached.

When the result is demanded by the parent process it has to wait until this result has been calculated. This locking can be realized easily: just create a process on the redex. A demanding process now automatically will have to wait until this process has delivered its normal form while it is guaranteed that the work is indeed being evaluated.

A subgraph is **self-contained** when the root of the subgraph is the only connection between the subgraph and the other parts of the graph. The rewriting of such a self-contained subgraph cannot be interfered with by rewriting actions performed elsewhere on other parts. Therefore it is safe to evaluate the redex *in parallel* with the other rewriting steps.

A divide-and-conquer example in FGRSs (Figure 14.2):

```
Fib 0   →   1
Fib 1   →   1
Fib n   →   + (LCopy left) (LCopy right),
            left:     {p!!} Fibd (– ({*} LCopy n) 1),
            right:    {p!!} Fibd (– ({*} LCopy n) 2)
```

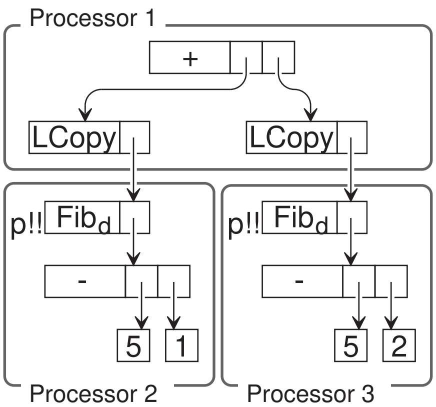


Figure 14.2 The process structure after one reduction of Fib 5. Each process may be executed in parallel on a different processor.

This particular combination of {p!!} annotations with the LCopy functions and defer indications makes it possible to evaluate both calls of Fib on another processor after which the result is copied back to the original demanding processor. The two process annotations indicate the reduction of the arguments with the functional strategy to normal form. The subgraph on which each process runs is made *self-contained* by actively taking copies (say n' and n") of the shared argument n. This lazy copy is made via the one step reduction {*}. So on one processor the graph Fib_d (– n' 1) is created, on the other Fib_d (– n'' 2). Furthermore, the parent process demands a lazy copy of the results to perform the addition, but the shipped redex is deferred and a process is created on it. So the parent process will wait until the shipped redex has been reduced to normal form, after which the lazy copy can be made.

14.3.2 Modelling loosely coupled evaluation

A method which makes it possible to model process behaviours that are more general than divide-and-conquer must provide a way to define *arbitrary* connections between processes and processors.

A lazy copy of a subgraph is generally *not* self-contained since it may contain lazy copy nodes that point to those parts in the graph that have to be copied later when they are in σ normal form and needed for the evaluation. So the lazy copy nodes form the border of the copy. Lazy copy nodes are natural candidates for serving as interconnections between parallel executing processes because they induce further copying when they are accessed. Hence, communication between parallel processes can be realized via lazy copy nodes. In this context lazy copy nodes are also called **communication channels**, or just **channels**, because these nodes will provide the communication between processors. The semantics of lazy copy nodes implies that the flow of data through a channel is the reverse of the direction of the node-id in the graph. Since channels are ordinary nodes, they can be passed as parameters or be copied as well.

A subgraph is **loosely connected** if channels (lazy copy nodes) are the only connections between the subgraph and the rest of the graph. So a self-contained subgraph is loosely connected if its root is a channel. Also, a loosely connected subgraph has the property that its nodes can be reduced *in parallel* when they are prohibited from being accessed by reducers running on other processes. Hence, a problem arises when a reducer demands information from a channel that points to a subgraph located on another processor. The root node of such a subgraph is called a **global node**. There are two possibilities:

- The demanded subgraph is in σ normal form. The information can be lazy copied to the demanding process, yielding again a loosely connected subgraph of which the copied parts can be reduced in parallel.
- The demanded subgraph is *not* in σ normal form. The demanding processor is not able to reduce the demanded subgraph when it is located on another processor. Therefore, the demanding process *has to be suspended* (*locked*) until the demanded information has been reduced to σ normal form by another process (on the corresponding processor). When the σ normal form is reached the sus-

pended process can be awoken and it can continue as explained in the previous case.

So processes running on different loosely connected subgraphs can run *in parallel* with each other *provided that these processes are locked when they demand information from a channel pointing to a subgraph which is not in* σ *normal form.* To make safe parallel evaluation possible this locking condition has to be fulfilled. This can be accomplished quite easily: always create a process on those subgraphs which channels of a loosely connected subgraph refer to. So processes are created on every deferred global node.

14.3.3 Translating {Par} and {Self}

In this subsection it is shown how annotations like {Par} and {Self} can be expressed in FGRSs using process annotations {p!!} and lazy copying.

To realize the parallel evaluation of the indicated function, with lazy copying a loosely connected subgraph is built. On this subgraph a parallel reducer {p!!} is created which reduces the spawned-off redex to normal form. The result has to be copied back lazily as well, so the root of the subgraph has to be created with the deferred attribute such that the result is not copied before the normal form is reached.

Each occurrence of:		will be substituted by:	
n :	{Par} Sym a_1 … a_m	n :	LCopy x,
		x :	{p!!} I_d ({*} LCopy y),
		y :	Sym a_1 … a_m
n :	{Self} Sym a_1 … a_m	n :	{p!!} Sym_d a_1 … a_m

I is the identity function: I x → x. The lazy copy created for the parallel process is loosely connected.

The semantics of {Self} just demands an interleaved executing process yielding a normal form for which the {p!!} annotation can be used. To prevent a subgraph being reduced by an interleaved reducer from being (lazy) copied while it is not yet in normal form, the subgraph is marked deferred as well. So indeed, all created processes (parallel as well as interleaved) are created on deferred subgraphs. In this way no lazy copies can be made of a subgraph under reduction. So {Par} and {Self} *processes act as copy-stoppers*. In fact, when this scheme is followed the only way to defer copying of a subgraph is by creating such a process on it. Hence, channels *always* refer to deferred subgraphs on which processes are created. The lazy copied subgraph on which a parallel process is created can have references to other parts somewhere in the original graph (see Figure 14.3). The copying of these parts is postponed because they were under reduction by another process at the time the lazy copy was made. Such a part will be lazy copied later to the created parallel process in the usual way when it is needed for evaluation and in normal form. Consequently, the process topologies that can be created with these annotations can become very complex.

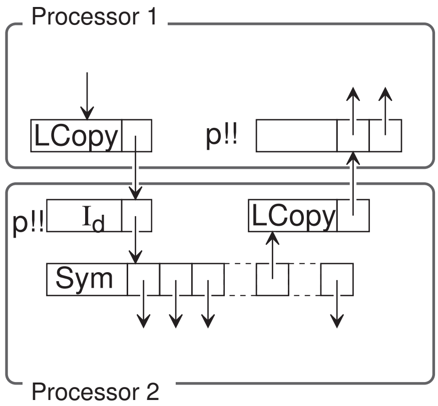


Figure 14.3 {Par} in terms of process annotations and lazy copying.

For instance, consider the following example using {Par} and {Self}:

Fib 0 → 1
Fib 1 → 1
Fib n → + left right,
left: {Par} Fib (– n 1),
right: {Self} Fib (– n 2)

This can be translated to:

Fib 0 → 1
Fib 1 → 1
Fib n → + (LCopy left) right,
left: {p$_{!!}$} I$_d$ ({*} LCopy y),
y: Fib (– n 1),
right: {p$_{!!}$} Fib$_d$ (– n 2)

The translation can be further optimized to:

Fib 0 → 1
Fib 1 → 1
Fib n → + (LCopy left) right,
left: {p$_{!!}$} Fib$_d$ (– {*} (LCopy n) 1),
right: {p$_{!!}$} Fib$_d$ (– n 2)

With the scheme above the semantics of {Par} and {Self} are *almost* completely expressed in terms of the primitives introduced in this chapter. There is one aspect which is different. In the scheme above a demanding reducer has to wait until the result is in normal form. We argued in Chapter 13 that to achieve optimal performance reducers should deliver their results as soon as possible. So one should not wait until the whole result is in normal form. Instead, that part of the graph that is already being calculated should be shipped. This means in the parallel case for normal form reducers that the deferred attribute should be taken away as soon as the root is in root normal form, while the deferred attribute should be added to the *arguments* of the node in root normal form to prevent them being copied as well before they have been reduced to root normal form.

There is another problem with normal form processes in general which also has to do with the demand to yield a result as soon as possible. Suppose that a normal form process yields an n-tuple. The functional strategy will force the evaluation of the elements of this tuple from left to right. But, in a parallel environment there may be a process demanding the result of the first element while another process may demand the result of the last element. It would be nice if this last process did not have to wait until all other tuple elements have been calculated. To make this possible one should create an (interleaved executing) process on any tuple element there is a demand for. One can also see that this is conceptually sound due to the decomposition property (see Chapter 13). If a tuple is known to be in PNF, so are the tuple elements (which means that it is sound to create a process on each of the tuple elements as well).

Concluding, {Par} and {Self} can be modelled as described above. However, an efficient implementation of normal form processes is rather complicated (see also Chapters 16 and 17).

14.3.4 Basic primitives for modelling process behaviour

The {Par} and {Self} annotations are high-level process annotations yielding a specific process behaviour suited for high-level functional programming. Their semantics is based on a strategy that reduces to normal form. However, in inter-process communication root normal forms play an essential role. So when process behaviours in general are investigated, it seems to be more natural to consider semantics based on processes that reduce to root normal form.

For that purpose in this section two new annotations are introduced: **{P}** for parallel processes and **{I}** for interleaved processes. The only difference with the {Par} and {Self} annotations is that root normal form processes {p!} are being used instead of normal form processes. The {P} and {I} annotations have the advantage that they create processes that deliver a result as soon as a root normal form is reached. Furthermore, these annotations have the advantage (and disadvantage) that one explicitly can (has to) specify what has to happen with this root normal form. Without special actions the whole subgraph in root normal form (possibly containing redexes) is lazily copied back to the demanding process. If the intention is only to ship the root in normal form one should explicitly start new processes (using either {P} or {I} annotations) on the arguments of this root node.

Each occurrence of:	will be substituted by:
n : {P} Sym $a_1 \dots a_m$	n : LCopy x, x : {$p_!$} I_d ({*} LCopy y), y : Sym $a_1 \dots a_m$
n : {I} Sym $a_1 \dots a_m$	n : {$p_!$} Sym_d $a_1 \dots a_m$

So in this scheme one has very fine control over what is shipped (work or data) and when (by creating a root normal form or not). This is handy when such fine tuning is needed. Take, for example, a function application that delivers a large structure after relatively few reduction steps. If a graph containing such a function application is submitted to another processor then it is preferable for the communication cost not to reduce this application before the submission. The fine control gets very annoying when one actually wants to ship a normal form. In that case the evaluation to normal form has to be programmed in the rewrite rules using root normal form processes.

The {P} and {I} annotations are interesting to study the behaviour of concurrent processes in functional languages due to the fine process tuning they offer (see Chapter 15). Furthermore, they have the advantage that they are relatively easy to implement (see Chapters 16 and 17).

Summary

- *Strategy annotations* are introduced in TGRSs. They make it possible to influence a single incarnation of a reduction strategy: the current strategy function is interrupted and the specified strategy function is applied on the indicated subgraph. When finally a σ normal form is reached, the interrupted strategy continues.
- *Process annotations* are introduced in TGRSs. They make it possible to change the reduction strategy into a parallel strategy with multiple incarnations of a reduction strategy: both the current strategy function and the new strategy function deliver candidates for rewriting. A reducer can be attached to each strategy function. So reducers are created dynamically and they die when a σ normal

form is reached. Parallel reduction is allowed under certain conditions only. In general, processes have to be *interleaved*.

- With *eager copying* a complete subgraph can be duplicated. Eager copying implies duplication of work.
- With *lazy copying*, one can copy a graph without duplicating work. This is realized by deferring the copy action until the graph that has to be copied is in σ normal form.
- With the lazy copy, primitive graph rewriting systems can be specified for which *the semantics are* in some sense *in between the semantics of term rewriting and of graph rewriting*. Subexpressions can be shared as well as copied. For certain redexes copying can be deferred until these redexes are in σ normal form.
- The use of lazy copying influences the resulting graph normal forms. The resulting graph normal form can even depend on the reduction order. But *the unravellings of the normal forms will always be equivalent*, which is required for term graph rewriting.
- A subgraph is *loosely connected* when lazy copy nodes (*channels*) are the only connections between the subgraph and the rest of the graph. Reduction on different loosely connected subgraphs can be done in *parallel*. Communication between loosely connected subgraphs can be modelled via *lazy copying*.
- The annotations `{Par}` and `{Self}` of the previous chapter can be modelled by using the proper combination of lazy copying with process and strategy annotations.
- More basic primitives are the introduced `{P}` and `{I}` annotations with semantics based on root normal form reduction. They can be used to specify various process behaviours (see the next chapter).

EXERCISES

14.1 Parallel reduction always has to be fair, but it may in practice be wise to reschedule certain reducers with a very low priority. Why?

14.2 Why is a deadlock in practice much more difficult to detect than a cycle-in-spine error?

14.3 A graph that has to be lazy copied to another machine will in practice probably be copied twice. Why?

14.4 Is the following type: `LCopy:: * -> UNQ *` correct? Would the `LCopy` function be useful as a primitive function for a Clean programmer? How about the `Copy` function? Which one do you prefer and why?