# Chapter 15
# Concurrent Clean


15.1 Extending Clean to Concurrent Clean
15.2 Specifying process structures
15.3 Translation into Concurrent Clean


This chapter extends the language Clean (see Chapter 8) with the possibility to create parallel and interleaved executing processes. The resulting language *Concurrent* Clean offers the {P} and {I} annotations as introduced in Chapter 14. A refined version of the {P} annotation is introduced as well. It can be used to give a more precise description of the processor on which a parallel process has to be allocated.

The Concurrent Clean annotations are explained in Section 15.1. How they can be used for the creation of various process topologies is explained in Section 15.2. The chapter is concluded with a discussion on how the {Par} and {Self} annotations of Chapter 13 can be translated into Clean's {P} and {I} annotations.

## 15.1 Extending Clean to Concurrent Clean

Concurrent Clean (Nöcker *et al.*, 1991b) incorporates a single extension with respect to Clean as defined in Chapter 8: *process annotations*. The **process annotations** of Concurrent Clean are precisely the {P} and {I} annotations presented in Chapter 14. There is an additional option to control the **process allocation**, i.e. one can specify with more or less precision on which concrete processor a new process has to be created.

The process annotations are *local* annotations (see Section 8.3) that can be specified on the right-hand side of a rewrite rule. They can be put on a node (before the symbol) or on a node-id. Just as with local strict annotations, it holds that if a local process annotation is put on a node, this is equivalent to putting the annotation on each reference to that node. Process annotations can have the form {P}, {I} or {P AT location} (see Appendix B). The semantics of {P} and {I} in Concurrent Clean fully conform with the definitions of {P} and {I} for parallel graph rewriting in Chapter 14. The effect of the annotations is summarized below. Furthermore, some explicit choices related to the order in which local and global annotations are effected in Clean are explained. The semantics of {P AT location} will be a small extension of the semantics of {P}.

### 15.1.1 Creation of processes on processors

When, in Concurrent Clean, a process annotation occurs in a contractum pattern, during a rewrite with the corresponding rule a new reducer will be created with as task the evaluation of the corresponding subgraph.

With the {I} annotation a new **interleaved reducer** is created on the *same* processor that reduces to root normal form (following the functional strategy). Such an interleaved reducer dies when the root normal form is reached. However, during the evaluation of this result other reducers may have been created.

With the {P} annotation a new **parallel reducer** is created. This reducer is *preferably* located on a *different* processor working on a lazy copy of the corresponding subgraph, but, when this is somehow not possible, it may be created on the *same* processor. Reducers that are located on different processors can be executed in parallel with each other. Reducers that run on the same processor run interleaved.

### 15.1.2 Locking

As explained before, an interleaved or parallel reducer will be *locked* if it wants to reduce a redex that is already being reduced by some other reducer. A locked reducer can continue when the redex has been reduced to root normal form. A reducer can demand the evaluation of a subgraph located on another processor. Such a demand always takes place via a *communication channel* (a lazy copy node). If the channel is referring to a subgraph in root normal form, a lazy copy of this subgraph is made on the processor such that it can be further evaluated by the demanding reducer. If the subgraph is not yet in root normal form, the demanding process is locked. In that case, on the other processor a reducer will always exist that reduces the subgraph to its root normal form. So all communication channels are always being *served* by processes to ensure that demanding processes are not locked unnecessarily.

### 15.1.3 Annotations and evaluation order

When there are several local annotations specified in a contractum, the *order* in which they have to be effected is, as is defined in the previous chapter, in principle depth-first with respect to the subgraph structure. The underlying philosophy here is that in many cases it makes sense to deal with the arguments before a function is called. In fact, the definitions of {P} and {I} in Chapter 14 also rely on this philosophy.

### *Local and global annotations*

An extra complication with respect to the semantics of annotations in Concurrent Clean is the fact that Clean (and hence Concurrent Clean) also allows strict annotations to appear in the type specification of a function, the *global* strict annotations (see Section 8.3.2). Consequently, it has to be defined in what order local and global annotations are effected. The general idea here is that local annotations are used by a programmer to control explicitly the order of evaluation of the right-hand side of a rewrite rule. Global annotations should not influence this explicit control. So the choice has been made to give priority to local annotations over global annotations. Of course, it still holds that when a chosen function is actually evaluated, first its globally annotated arguments will be reduced.

### *Maximizing actual parallelism*

When there is not a clear hierarchy between annotations, it would be nice if the degree of parallelism would be maximized in choosing between the various possibilities.

So for these cases the choice has been made first to effect {P}s, then {I}s, and finally {!}s. These priorities also hold for the cyclic case and for the case in which several annotations are put on the same node. At first sight it may not seem to make much sense even to allow the latter case because only one reducer will be able to reduce the indicated subgraph. All others will be locked until the reduction is finished. However, a user may want to specify such a behaviour in which, for example, first a parallel reducer is started on a node (with a {P}) and immediately the parent process will wait for the result (due to a {!}).

However, parallelism can be even further maximized, as is clarified by the following situation: suppose that a local process annotation and a global strict annotation are both valid for the same node. Then, as defined above, the priority lies with creating the child process first in order to maximize the parallelism. It can, however, be the case that the parent process has again the choice between waiting for the child process to produce its result and, for instance, evaluating another argument of the corresponding function call. Then the parallel behaviour of the program is improved when the other argument is chosen first. Otherwise, the parent process would just be waiting in parallel.

So when arguments are equal in priority (e.g. they are either both globally strict annotated or both locally strict annotated) the choice is made to effect first those strict annotations for which it is known that (owing to an analysis of the right-hand side of the corresponding rule) yet another process is not already evaluating the corresponding subgraph. In this way unintended waiting is avoided.

Example of maximizing parallelism:

```
:: Fib INT    ->   INT;
   Fib n      ->   + ({P} Fib (– n 1)) (Fib (– n 2));
```

+ is globally strict in both arguments. A child process reduces a lazy copy of the first argument in parallel. The parent continues with the second argument.

```
   Fib n      ->   + (Fib (– n 1)) ({P} Fib (– n 2));
```

But if a child were to be created on the second argument instead, the parent process would evaluate the first argument. In this way a nice symmetric behaviour is obtained, with the maximum amount of actual parallelism.

### 15.1.4 Controlling process allocation

Processors in a distributed architecture are generally sparsely connected. When two parallel processes communicate heavily with each other it is efficient to locate them on processors that are directly connected via a hardware link. If one is aiming for optimal performance the actual location of a process should be definable. For this reason there is a possibility in Concurrent Clean to specify the processor on which a parallel process should be created. The use of this possibility might make the efficiency of the program fully dependent on the actual machine the program is running on. So the user is advised to refrain from the use of this feature as much as possible.

The {P} annotation can be extended with a **location directive** {P AT location}, where location is an expression of predefined type PROCID indicating the processor on which the parallel process has to be created. In the library deltaP (see Appendix B) functions are given that yield an object of this type. Some of the primitives can be used without detailed knowledge of the actual processor topology. For instance, with the function NeighbourP one can create a process *n* links (hops) away from another processor (e.g. the current processor CurrentP). With the primitive ITOP one can specify exactly which processor is intended. The primitive converts an integer value to a value of type PROCID. It is assumed that each processor of a concrete architecture can be uniquely identified by an integer value. How this mapping is defined is implementation-dependent. So the programmer must know this mapping and the actual processor topology to make optimal use of it.

In any case, it holds that whatever the use of process locations is, the *result of the program cannot be influenced*. The process locations influence only the actual process allocation on the machine.

## 15.2 Specifying process structures

The fact that Concurrent Clean is a graph rewriting language is clearly an advantage for the specification of process topologies. The graph structure can be used to define explicitly the demanded process topology. This makes, for instance, the creation of cyclic process structures much better to understand. To illustrate the use of graph rewriting in specifying process topologies, several Concurrent Clean examples are given below.

### 15.2.1 A hierarchical process topology

Divide-and-conquer parallelism is expressed with hierarchical process topologies. They have a straightforward process behaviour. This can be expressed very easily in Concurrent Clean.

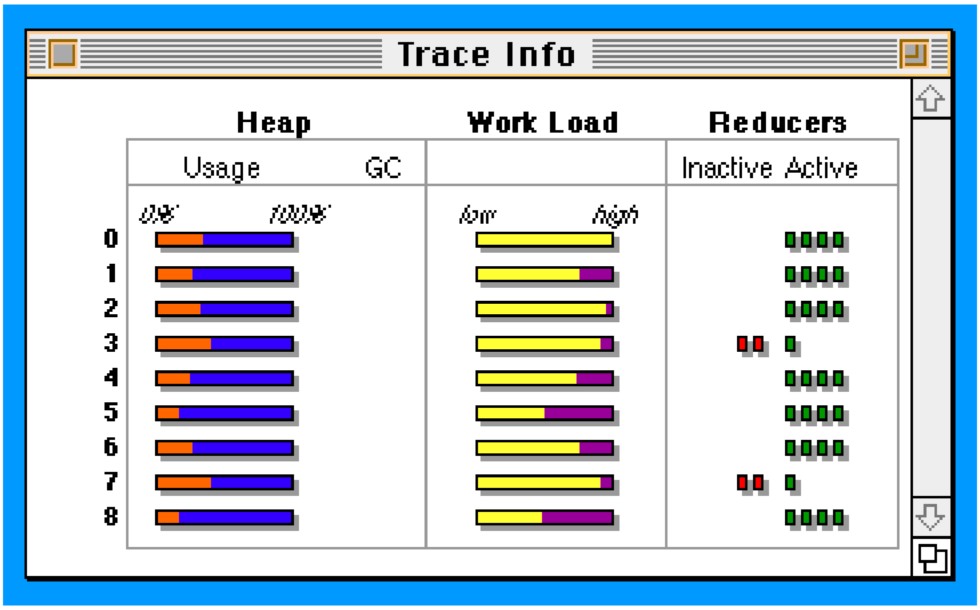
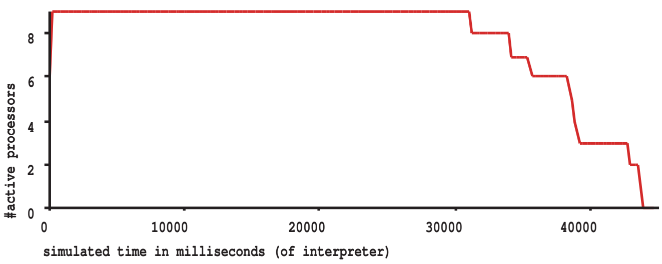



**Figure 15.1** Trace information shown by the Concurrent Clean simulator. For each simulated processor the heap usage is displayed, along with the workload relative to the other processors as well as the number of inactive (locked) and active reducers.




| | |
|---|---|
| **Speedup fib** | **776 %** |
| #processors | 9 |
| #processes | 34 |
| #suspensions | 33 |
| communication | 2244 bytes |

**Figure 15.2** During simulation all kinds of statistical information can be written to a file and examined afterwards.

A hierarchical process topology in Concurrent Clean.

```
:: Fib INT  ->   INT;
   Fib 0    ->   1;
   Fib 1    ->   1;
   Fib n    ->   + {P} left {P} right,      IF  > n Threshold
            ->   + left right,              IF  > n 2
            ->   ABORT "Fib called with negative value",
                 left:     Fib ! (– n 1),
                 right:    Fib ! (– n 2);
```

Both calls of Fib can be evaluated in parallel on another processor. First (– n 1) and (– n 2) are evaluated, forced by the local strict annotations because these annotations are put on subgraphs of the graphs on which the process annotations are put. Next, two parallel processes are created on lazy copies of right and left respectively. The parent process continues with the addition, but it has to wait until the arguments are evaluated and have been lazy copied back. A more efficient solution would be to create a child process for just one of the arguments. The parent process then will automatically calculate the other. This version has been used in Figures 15.1 and 15.2.

The Concurrent Clean system provides tools to simulate the behaviour of parallel programs, using either the interpreter or the code generator. In Figure 15.1 a snapshot is given of the trace information displayed by the interpreter simulating Fib 22 on 9 processors. Also statistical information can be produced and examined after execution (see Figure 15.2).

### 15.2.2 A simple non-hierarchical process topology

In the following example a simple non-hierarchical parallel reducer topology is demonstrated. It serves the purpose of explaining how more complicated parallel reducer topologies can be expressed.

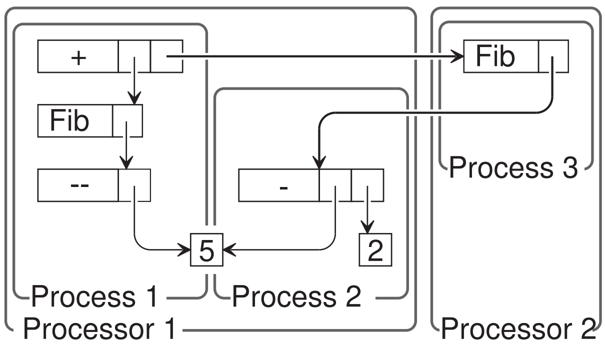



**Figure 15.3** Intended distribution of the processes across the processors. A snapshot of the program execution of Fib 5 is given.

A non-hierarchical parallel reducer structure: the second argument of Fib will be executed by another parallel reducer, but the argument of that call is reduced interleaved on the original processor (see Figure 15.3).

```
:: Fib INT  ->  INT;
   Fib 0    ->  1;
   Fib 1    ->  1;
   Fib n    ->  + (Fib (-- n)) m,
                m:   {P} Fib o,
                o:   {I} – n 2;
```

### 15.2.3 Asynchronous process communication with streams

The small examples above are very similar to the examples using {Par} and {Self} since only simple basic values are communicated. For graphs yielding basic values the root normal form is also the normal form.

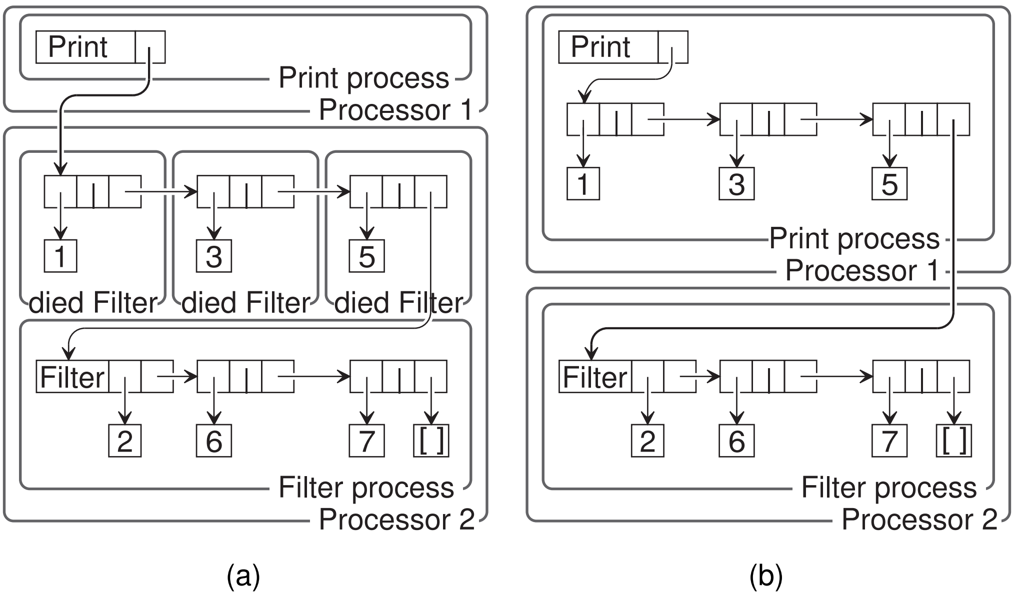



**Figure 15.4** (a) Snapshot of the filtering process working on a list containing the natural numbers from 1 to 7. (b) The root normal forms yielded by successive interleaved Filter processes have been shipped with one lazy copy action. The active interleaved Filter process acts as a copy-stopper and a new channel has been created to this process. If more results are demanded, the lazy copy action will be continued as soon as new results become available.

To express a general asynchronous communication stream a sequence of interleaved reducers has to be created. Each interleaved reducer computes a partial result that can be sent across a channel. Just before the reducer delivers its root normal form (and dies) it creates a new interleaved reducer that delivers the next result in the same way. When partial results are delivered more frequently than is demanded, all partial results (a **burst**) are sent at the same time. The lazy copy will copy all partial results at once, but it stops at those nodes that are under reduction by a reducer. In this way a parallel reducer can have several streams and all kinds of (cyclic) graphs can be sent to another processor.

Example of an asynchronous communication behaviour with streams:

```
:: PrintList [INT]          ->   [CHAR];
   PrintList list           ->   Print ({P} Filter 2 list);

:: Filter    INT   [INT]    ->   [INT];
   Filter    p     [ ]      ->   [ ];
   Filter    p     [ f | r ] ->  Filter p r,          IF  = (% f p) 0
                            ->   [ f | {I} Filter p r ];
```

The process reducing PrintList creates a new parallel reducer: Filter. This process removes from its second argument, which is a list, all the elements that are divisible by the number 2. When Filter has computed an element of the result list it creates a new Filter process to compute the next number, after which it dies. Several results may be added to the stream before they are communicated on demand. The end of the stream will be a channel referring to the current filtering process (see Figure 15.4).

The sieve of Eratosthenes is a classical example in which parallel sieving processes are created dynamically in a pipeline (see also Section 13.3.1). In Concurrent Clean the sieve program is expressed as follows (Limit is assumed to be defined in a MACRO, Filter is defined above):

```
:: Start ->    [INT];
   Start ->    Primes;

:: Primes  ->    [INT];
   Primes  ->    Sieve ({P} Gen 2);

:: Gen INT    ->    [INT];
   Gen Limit  ->    [ ];
   Gen n      ->    [ n | {I} Gen (++ n) ];

:: Sieve [INT]     ->    [INT];
   Sieve [ ]       ->    [ ];
   Sieve [ p | s ] ->    [ p | {P} Sieve ({I} Filter p s) ];
```

### 15.2.4 Communicating work or data

When a reducer yields a root normal form it dies and the yielded subgraph in root normal form can be lazy copied to a demanding process.

The copying process will postpone copying on those parts of the graph that are under reduction by another process. This makes it possible to decide which part of the graph is lazy copied back. The subgraph in root normal form will generally contain several redexes (work). When such a redex is returned in the lazy copy, work is shipped from one processor to another. One can choose whether or not work should be locally calculated before it is shipped. In that case an interleaved process should be created on the corresponding redex before the root normal form is delivered. So for each redex one can choose how it should be handled.

Choosing between data or work:

```
:: F  (=> INT INT)  BOOL   ->    [ INT ];
   F  g             h      ->    Map g ({P} BigList h);

:: BigList  BOOL  ->   [ INT ];
   BigList  h     ->   [ 0 | Gen 2 ],   IF  h
                  ->   [ 1 | {I} CompList ];
```

The function F maps a function g over a list that is computed by a parallel process BigList (Map is defined in Section 8.1.6). Depending on the evaluation of h, either a simple big list is returned or a big list that takes some computing power. In the first case, work is copied back: the list itself is not returned but an expression is returned that computes this list when it is evaluated. In the second case data is copied, computed by an interleaved process CompList that is created to compute the list before it is shipped.

### 15.2.5 Cyclic process topologies with location directives

The following example creates a cyclic parallel process structure, i.e. a number of parallel reducers that are mutually dependent. In a similar way cyclic process structures can be explicitly specified in more elaborate examples, such as the Warshall example of Chapter 13.

```
:: Start ->     [INT];
   Start ->     last:  CreateProcs NrOfProcs last;

:: NrOfProcs ->    INT;
   NrOfProcs ->    5;

:: CreateProcs INT [INT]    ->    [INT];
   CreateProcs 1   left     ->    Process 1 left;
   CreateProcs pid left     ->    CreateProcs (– – pid)
                                                   ({P} Process pid left);
:: Process INT   [INT]    ->    [INT];
   Process n     left     ->    [ n | {I} Process (+ n (Hd left)) (Tl left) ];
```

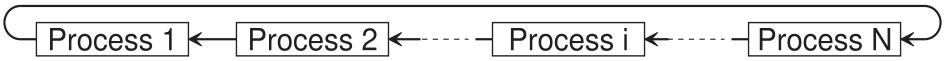



**Figure 15.5** A cyclic process structure.

CreateProcs is initially called with a reference to itself, thus forming a cyclic process structure of length one. CreateProcs recursively creates new processes Process. Each Process has a process number. CreateProcs will finally form the first Process in the cycle with process number 1. Each Process is connected to the next one, i.e. the one with the next pid number, by means of a channel (see Figure 15.5). During the creation of the processes this channel is passed as a parameter: left. Hd and Tl take, in a lazy way, respectively the head and the tail of the channel. Note that the type of left has to be the same as the result type of CreateProcs and Process. The function Process will in general produce a lazy stream of values (in the example integer values) that can be used as input by another Process. Owing to the use of {I} annotations recursively a part of the stream can be produced as a result. In the example an infinite list of rapidly increasing numbers is generated ([1, 3, 8, 20, 48, …]).

As explained in Chapter 14, one has to be careful not to introduce deadlock when cyclic process topologies are specified using recursive functions. A subtle deadlock example is given by a variant of the function definition Process of the cyclic process topology example above.

```
:: Process INT   [INT]       ->   [INT];
   Process n     [ hd | tl ] ->   [ n | {I} Process (+ n hd) tl ];
```

A deadlock occurs since each process, due to the pattern match, wants the result of the previous process before it creates a result itself.

### *Location directives*

In the cyclic example given above every process is communicating heavily with both its neighbours. So it is probably more efficient to create a new process on a processor to which a direct link exists. This can be done by replacing the {P} annotations above by the location directive {P AT NeighbourP 1 CurrentP}. However, this only guarantees that the new process will be located on a direct neighbour: it can happen that effectively only two processors are used. This can be avoided by making use of knowledge about the actual machine topology, e.g. using the location function ITOP (see also Appendix B).

A cycle of processes assuming that a neighbouring processor has an identification one lower than the previous one.

```
CreateProcs p l -> CreateProcs (– – p) ({P AT ITOP p} Process p l);
```

## 15.3 Translation into Concurrent Clean

In many cases it is possible to translate {Par} and {Self} annotations to a collection of {P}, {I} and {!} annotations. When root normal forms are equivalent to normal forms (e.g. for simple basic values) the translation is trivial. When this is not the case, a sequence of annotations is needed to achieve the evaluation to normal form. A process reducing to normal form can be realized by creation of a new interleaved process just before the current one delivers its root normal form. Compare for instance the sieve example in this chapter with the sieve example in Chapter 13. The creation of additional interleaved processes has to be explicitly specified in the rewrite rules. Consequently, it is not possible to use this translation scheme to force normal form evaluation of an arbitrary function passed as an argument to a function.

Limitation of simulating normal form processes via a sequence of root normal form processes:

```
:: ParMap (=> x y) [x] -> [y];
   ParMap f [ ]          ->    [ ];
   ParMap f [ x | xs ]   ->    [ {P} f x | {I} ParMap f xs ];
```

The parallel process that is created in the ParMap example only evaluates the unknown function application f x to *root* normal form. It is not possible to force the evaluation of such an unknown function to normal form.

So the ability to create processes that reduce arbitrary unknown functions to normal form gives more expressive power to the language. As already stated in Chapter 13, normal form processes are often useful, but also hard to implement efficiently. Process types (see Chapter 13.2.5) can help to set up efficient communication streams between normal form processes.

The {P} and {I} annotations have the advantage that they allow a more precise control over the interaction between processes. So they are also very useful primitives. They are less suited when evaluation to normal form is demanded.

In future versions of Concurrent Clean normal form processes with process types will be added in addition to the current annotations. But it remains to be seen what kind of annotations in the long term are both suitable from a programming point of view as well as efficiently implementable on a wide range of architectures. A different approach can be found in Darlington *et al*. (1991), who propose a wide range of special purpose annotations (called skeletons) for all kinds of architectures (varying from MIMD to SIMD machines).

## *Summary*

- Concurrent Clean extends Clean with *process annotations* {P}, {I} and {P AT location}. Their semantics *conforms* fully to *the semantics of parallel graph rewriting* as given in Chapter 14.
- With the {I} annotation a new *interleaved reducer* is created on the *same* processor reducing to root normal form following the functional strategy. With the {P} annotation a *parallel reducer* is created *preferably* running on a *different* processor. Communication takes place via lazy copying, as explained in the previous chapter.
- A *location directive* can be added to indicate on which processor a process should be created. Location directives cannot influence the result of the program.
- It is not always possible to translate {Par} and {Self} annotations to a collection of {P}, {I} and {!} annotations. Both kinds of annotation are useful. {P} and {I} give precise control over the distribution of work or data and are efficiently implementable because they fit very well in the default evaluation order (which is root normal form evaluation). {Par} and {Self} are much more difficult to implement efficiently but they are user friendly because processes are in practice often created to yield a normal form.

## EXERCISES

**15.1** What other kind of location directives can you imagine as being useful? Explain why.

**15.2** Rewrite Exercises 13.2 and 13.3 in Concurrent Clean. Simulate parallelism using the simulator on 1, 2, 4, 8 and 16 processors. Measure the speed-ups. Explain the results.

**15.3** Answer the questions of Exercise 15.2 for the sieve example given in Section 15.2.3.

**15.4** Would it not be better for parallel evaluation to have an eager language instead of a lazy one?

**15.5** Can branch-and-bound parallelism be expressed in a pure functional language?