

**Memory management** There are two modifications to memory management:

- Extending the definition of reachability: A variable  $x$  is reachable if the trigger store contains  $trig(x, y)$  and  $y$  is reachable.
- Reclaiming triggers: If a variable  $y$  becomes unreachable and the trigger store contains  $trig(x, y)$ , then remove the trigger.

### Needing a variable

What does it mean for a variable to be needed? The definition of *need* is carefully designed so that lazy execution is declarative, i.e., all executions lead to logically-equivalent stores. A variable is *needed* by a suspended operation if the variable must be determined for the operation to continue. Here is an example:

```
thread X={ByNeed fun {$} 3 end} end
thread Y={ByNeed fun {$} 4 end} end
thread Z=X+Y end
```

To keep the example simple, let us consider that each thread executes atomically. This means there are six possible executions. For lazy execution to be declarative, all of these executions must lead to equivalent stores. Is this true? Yes, it is true, because the addition will wait until the other two triggers are created, and these triggers will then be activated.

There is a second way a variable can be needed. A variable is *needed* if it is determined. If this were not true, then the demand-driven concurrent model would not be declarative. Here is an example:

```
thread X={ByNeed fun {$} 3 end} end
thread X=2 end
thread Z=X+4 end
```

The correct behavior is that *all* executions should fail. If  $x=2$  executes last then the trigger has already been activated, binding  $x$  to 3, so this is clear. But if  $x=2$  is executed *first* then the trigger should also be activated.

Let us conclude by giving a more subtle example:

```
thread X={ByNeed fun {$} 3 end} end
thread X=Y end
thread if X==Y then Z=10 end end
```

Should the comparison  $x==y$  activate the trigger on  $x$ ? According to our definition the answer is *no*. If  $x$  is made determined then the comparison will still not execute (since  $y$  is unbound). It is only later on, if  $y$  is made determined, that the trigger on  $x$  should be activated.

Being needed is a *monotonic* property of a variable. Once a variable is needed, it stays needed forever. Figure 4.25 shows the stages in a variable's lifetime. Note that a determined variable is always needed, just by the fact of being determined. Monotonicity of the need property is essential to prove that the demand-driven concurrent model is declarative.

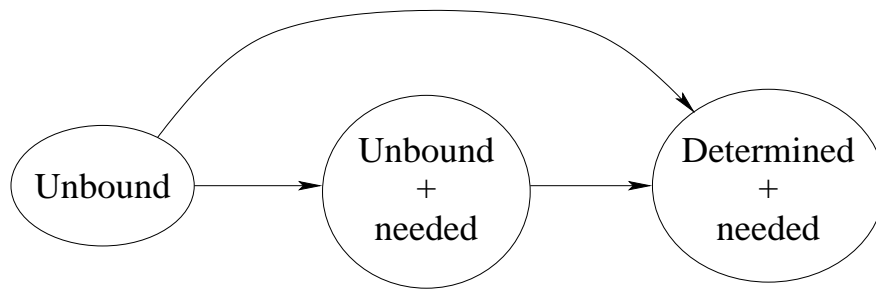


Figure 4.25: Stages in a variable's lifetime

### Using by-need triggers

By-need triggers can be used to implement other concepts that have some “lazy” or “demand-driven” behavior. For example, they underlie lazy functions and dynamic linking. Let us examine each in turn.

**Implementing lazy functions with by-need** A lazy function is evaluated only when its result is needed. For example, the following function generates a lazy list of integers:

```
fun lazy {Generate N} N|{Generate N+1} end
```

This is a linguistic abstraction that is defined in terms of `ByNeed`. It is called like a regular function:

```
L={Generate 0}
{Browse L}
```

This will display nothing until `L` is needed. Let us ask for the third element of `L`:

```
{Browse L.2.2.1}
```

This will calculate the third element, 2, and then display it. The linguistic abstraction is translated into the following code that uses `ByNeed`:

```
fun {Generate N}
  {ByNeed fun {$} N|{Generate N+1} end}
end
```

This uses procedural abstraction to delay the execution of the function body. The body is packaged into a zero-argument function which is only called when the value of `{Generate N}` is needed. It is easy to see that this works for all lazy functions. Threads are cheap enough in Mozart that this definition of lazy execution is practical.

**Implementing dynamic linking with by-need** We briefly explain what dynamic linking is all about and the role played by lazy execution. Dynamic linking is used to implement a general approach to structuring applications called component-based programming. This approach was introduced in Section 3.9

and is explained fully in Chapters 5 and 6. Briefly, an application’s source code consists of a set of component specifications, called *functors*. A running application consists of instantiated components, called *modules*. A module is represented by a record that groups together the module’s operations. Each record field references one operation. Components are *linked* when they are needed, i.e., their functors are loaded into memory and instantiated. As long as the module is not needed, then the component is not linked. When a program attempts to access a module field, then the component is needed and by-need execution is used to link the component.

### 4.5.2 Declarative computation models

At this point, we have defined a computation model with both laziness and concurrency. It is important to realize that these are independent concepts. Concurrency can make batch computations incremental. Laziness can reduce the amount of computation needed to get a result. A language can have neither, either, or both of these concepts. For example, a language with laziness but no concurrency does coroutines between a producer and a consumer.

Let us now give an overview of all the declarative computation models we know. All together, we have added three concepts to strict functional programming that preserve declarativeness while increasing expressiveness: dataflow variables, declarative concurrency, and laziness. Adding these concepts in various combinations gives six different practical computation models, as summarized in Figure 4.26.<sup>11</sup> Dataflow variables are a prerequisite for declarative concurrency, since they are the mechanism by which threads synchronize and communicate. However, a sequential language, like the model of Chapter 2, can also have dataflow variables and use them to good effect.

Since laziness and dataflow variables are independent concepts, this means there are *three special moments* in a variable’s lifetime:

1. Creation of the variable as an entity in the language, such that it can be placed inside data structures and passed to or from a function or procedure. The variable is not yet bound to its value. We call such a variable a “dataflow variable”.
2. Specification of the function or procedure call that will evaluate the value of the variable (but the evaluation is not done yet).
3. Evaluation of the function. When the result is available, it is bound to the variable. The evaluation might be done according to a trigger, which may be implicit such as a “need” for the value. Lazy execution uses implicit need.

---

<sup>11</sup>This diagram leaves out *search*, which leads to another kind of declarative programming called relational programming. This is explained in Chapter 9.

	<i>sequential with values</i>	<i>sequential with values and dataflow variables</i>	<i>concurrent with values and dataflow variables</i>
<i>eager execution (strictness)</i>	strict functional programming (e.g., Scheme, ML) (1)&(2)&(3)	declarative model (e.g., Chapter 2, Prolog) (1), (2)&(3)	data-driven concurrent model (e.g., Section 4.1) (1), (2)&(3)
<i>lazy execution</i>	lazy functional programming (e.g., Haskell) (1)&(2), (3)	lazy FP with dataflow variables (1), (2), (3)	demand-driven concurrent model (e.g., Section 4.5.1) (1), (2), (3)

(1): Declare a variable in the store

(2): Specify the function to calculate the variable's value

(3): Evaluate the function and bind the variable

(1)&(2)&(3): Declaring, specifying, and evaluating all coincide

(1)&(2), (3): Declaring and specifying coincide; evaluating is done later

(1), (2)&(3): Declaring is done first; specifying and evaluating are done later and coincide

(1), (2), (3): Declaring, specifying, and evaluating are done separately

Figure 4.26: Practical declarative computation models

These three moments can be done separately or at the same time. Different languages enforce different possibilities. This gives four variant models in all. Figure 4.26 lists these models, as well as the two additional models that result when concurrency is added as well. For each of the variants, we show an example with a variable `X` that will eventually be bound to the result of the computation `11*11`. Here are the models:

- In a strict functional language with values, such as Scheme or Standard ML, moments (1) & (2) & (3) must always coincide. This is the model of Section 2.7.1. For example:

```
declare X=11*11                                % (1)+(2)+(3) together
```

- In a lazy functional language with values, such as Haskell, moments (1) & (2) always coincide, but (3) may be separate. For example (defining first a lazy function):

```
declare fun lazy {LazyMul A B} A*B end
declare X={LazyMul 11 11}    % (1)+(2) together
{Wait X}                    % (3) separate
```

This can also be written as:

```
declare X={fun lazy {$} 11*11 end}    % (1)+(2) together
{Wait X}                               % (3) separate
```

- In a strict language with dataflow variables, moment (1) may be separate and (2) & (3) always coincide. This is the declarative model, which is defined in Chapter 2. This is also used in logic programming languages such as Prolog. For example:

```
declare X                                % (1) separate
X=11*11                                  % (2)+(3) together
```

If concurrency is added, this gives the data-driven concurrent model defined at the beginning of this chapter. This is used in concurrent logic programming languages. For example:

```
declare X                                % (1) separate
thread X=11*11 end                      % (2)+(3) together
thread if X>100 then {Browse big} end end % Conditional
```

Because dataflow variables are single-assignment, the conditional always gives the same result.

- In the demand-driven concurrent model of this chapter, moments (1), (2), (3) may all be separate. For example:

```
declare X                                % (1) separate
X={fun lazy {$} 11*11 end}              % (2) separate
{Wait X}                                 % (3) separate
```

When concurrency is used explicitly, this gives:

```
declare X                                % (1)
thread X={fun lazy {$} 11*11 end} end % (2)
thread {Wait X} end                     % (3)
```

This is the most general variant model. The only connection between the three moments is that they act on the same variable. The execution of (2) and (3) is concurrent, with an implicit synchronization between (2) and (3): (3) waits until (2) has defined the function.

In all these examples, *x* is eventually bound to 121. Allowing the three moments to be separate gives maximum expressiveness within a declarative framework. For example, laziness allows to do declarative calculations with potentially infinite lists. Laziness allows to implement many data structures as efficiently as with explicit state, yet still declaratively (see, e.g., [138]). Dataflow variables allow to write concurrent programs that are still declarative. Using both together allows to write concurrent programs that consist of stream objects communicating through potentially infinite streams.

One way to understand the added expressiveness is to realize that dataflow variables and laziness each add a weak form of state to the model. In both cases, restrictions on using the state ensure the model is still declarative.

### Why laziness with dataflow must be concurrent

In a functional language without dataflow variables, laziness can be sequential. In other words, demand-driven arguments to a lazy function can be evaluated sequentially (i.e., using coroutines). If dataflow variables are added, this is no longer the case. A deadlock can occur if the arguments are evaluated sequentially. To solve the problem, the arguments must be evaluated concurrently. Here is an example:

```

local
  Z
  fun lazy {F1 X}      X+Z end
  fun lazy {F2 Y} Z=1 Y+Z end
in
  {Browse {F1 1}+{F2 2}}
end

```

This defines `F1` and `F2` as lazy functions. Executing this fragment displays 5 (do you see why?). If `{F1 1}` and `{F2 2}` were executed sequentially instead of concurrently, then this fragment would deadlock. This is because `X+Z` would block and `Z=1` would never be reached. A question for the astute reader: which of the models in Figure 4.26 has this problem? The binding of `Z` done by `F2` is a kind of “declarative side effect”, since `F2` changes its surroundings through a means separate from its arguments. Declarative side effects are usually benign.

It is important to remember that a language with dataflow variables and concurrent laziness is still declarative. There is no observable nondeterminism. `{F1 1}+{F2 2}` always gives the same result.

### 4.5.3 Lazy streams

In the producer/consumer example of Section 4.3.1, it is the producer that decides how many list elements to generate, i.e., execution is eager. This is a reasonable technique if the total amount of work is finite and does not use many system resources (e.g., memory or processor time). On the other hand, if the total work potentially uses many resources, then it may be better to use lazy execution. With lazy execution, the consumer decides how many list elements to generate. If an extremely large or a potentially unbounded number of list elements are needed, then lazy execution will use many fewer system resources at any given point in time. Problems that are impractical with eager execution can become practical with lazy execution. On the other hand, lazy execution may use many more *total* resources, because of the cost of its implementation. The need for laziness must take both of these factors into account.

Lazy execution can be implemented in two ways in the declarative concurrent model: with *programmed* triggers or with *internal* triggers. Section 4.3.3 gives an example with programmed triggers. Programmed triggers require explicit communications from the consumer to the producer. A simpler way is to use

internal triggers, i.e., for the language to support laziness directly. In that case the language semantics ensures that a function is evaluated only if its result is needed. This makes the function definition simpler because it does not have to do the “bookkeeping” of the trigger messages. In the demand-driven concurrent model we give syntactic support to this technique: the function can be annotated as “lazy”. Here is how to do the previous example with a lazy function that generates a potentially infinite list:

```

fun lazy {Generate N}
  N|{Generate N+1}
end

fun {Sum Xs A Limit}
  if Limit>0 then
    case Xs of X|Xr then
      {Sum Xr A+X Limit-1}
    end
  else A end
end

local Xs S in
  Xs={Generate 0}           % Producer
  S={Sum Xs 0 150000} % Consumer
  {Browse S}
end

```

As before, this displays 11249925000. Note that the `Generate` call does *not* need to be put in its own thread, in contrast to the eager version. This is because `Generate` creates a by-need trigger and then completes.

In this example, it is the consumer that decides how many list elements should be generated. With eager execution it was the producer that decided. In the consumer, it is the **case** statement that needs a list pair, so it implicitly triggers the generation of a new list element `x`. To see the difference in resource consumption between this version and the preceding version, try both with 150000 and then with 15000000 elements. With 150000 elements, there are no memory problems (on a personal computer with 64MB memory) and the eager version is faster. This is because of the overhead of the lazy version’s implicit triggering mechanism. With 15000000 elements, the lazy version needs only a very small memory space during execution, while the eager version needs a huge memory space. Lazy execution is implemented with the `ByNeed` operation (see Section 4.5.1).

### Declaring lazy functions

In lazy functional languages, *all* functions are lazy by default. In contrast to this, the demand-driven concurrent model requires laziness to be declared explicitly, with the `lazy` annotation. We find that this makes things simpler both for the programmer and the compiler, in several ways. The first way has to do with efficiency and compilation. Eager evaluation is several times more efficient

than lazy evaluation because there is no triggering mechanism. To get good performance in a lazy functional language, this implies that the compiler has to determine which functions can safely be implemented with eager evaluation. This is called *strictness analysis*. The second way has to do with language design. An eager language is much easier to extend with non-declarative concepts, e.g., exceptions and state, than a lazy language.

### Multiple readers

The multiple reader example of Section 4.3.1 will also work with lazy execution. For example, here are three lazy consumers using the `Generate` and `Sum` functions defined in the previous section:

```
local Xs S1 S2 S3 in
  Xs={Generate 0}
  thread S1={Sum Xs 0 150000} end
  thread S2={Sum Xs 0 100000} end
  thread S3={Sum Xs 0 50000} end
end
```

Each consumer thread asks for stream elements independently of the others. If one consumer is faster than the others, then the others may not have to ask for the stream elements, if they have already been calculated.

#### 4.5.4 Bounded buffer

In the previous section we built a bounded buffer for eager streams by explicitly programming the laziness. Let us now build a bounded buffer using the laziness of the computation model. Our bounded buffer will take a lazy input stream and return a lazy output stream.

Defining a lazy bounded buffer is a good exercise in lazy programming because it shows how lazy execution and data-driven concurrency interact. Let us do the design in stages. We first specify its behavior. When the buffer is first called, it fills itself with  $n$  elements by asking the producer. Afterwards, whenever the consumer asks for an element, the buffer in its turn asks the producer for another element. In this way, the buffer always contains up to  $n$  elements. Figure 4.27 shows the resulting definition. The call `{List.drop In N}` skips over  $N$  elements of the stream `In`, giving the stream `End`. This means that `End` always “looks ahead”  $N$  elements with respect to `In`. The lazy function `Loop` is iterated whenever a stream element is needed. It returns the next element `!r` but also asks the producer for one more element, by calling `End.2`. In this way, the buffer always contains up to  $N$  elements.

However, the buffer of Figure 4.27 is incorrect. The major problem is due to the way lazy execution works: the calculation that needs the result will block while the result is being calculated. This means that when the buffer is first



```

fun {Buffer1 In N}
  End={List.drop In N}
  fun lazy {Loop In End}
    case In of I|In2 then
      I|{Loop In2 End.2}
    end
  end
in
  {Loop In End}
end

```

Figure 4.27: Bounded buffer (naive lazy version)

```

fun {Buffer2 In N}
  End=thread {List.drop In N} end
  fun lazy {Loop In End}
    case In of I|In2 then
      I|{Loop In2 thread End.2 end}
    end
  end
in
  {Loop In End}
end

```

Figure 4.28: Bounded buffer (correct lazy version)

called, it cannot serve any consumer requests until the producer generates  $n$  elements. Furthermore, whenever the buffer serves a consumer request, it cannot give an answer until the producer has generated the next element. This is too much synchronization: it links together the producer and consumer in lock step! A usable buffer should on the contrary *decouple* the producer and consumer. Consumer requests should be serviced whenever the buffer is nonempty, independent of the producer.

It is not difficult to fix this problem. In the definition of `Buffer1`, there are two places where producer requests are generated: in the call to `List.drop` and in the operation `End.2`. Putting a `thread ... end` in both places solves the problem. Figure 4.28 shows the fixed definition.

### Example execution

Let us see how this buffer works. We define a producer that generates an infinite list of successive integers, but only one integer per second:

```

fun lazy {Ints N}
  {Delay 1000}
  N|{Ints N+1}
end

```

Now let us create this list and add a buffer of 5 elements:

```
declare
In={Ints 1}
Out={Buffer2 In 5}
{Browse Out}
{Browse Out.1}
```

The call `Out.1` requests one element. Calculating this element takes one second. Therefore, the browser first displays `Out<Future>` and one second later adds the first element, which updates the display to `1|_<Future>`. The notation “`_<Future>`” denotes a read-only variable. In the case of lazy execution, this variable has an internal trigger attached to it. Now wait at least 5 seconds, to let the buffer fill up. Then enter:

```
{Browse Out.2.2.2.2.2.2.2.2.2.2}
```

This requests 10 elements. Because the buffer only has 5 elements, it is immediately emptied, displaying:

```
1|2|3|4|5|6|_<Future>
```

One more element is added each second for four seconds. The final result is:

```
1|2|3|4|5|6|7|8|9|10|_<Future>
```

At this point, all consumer requests are satisfied and the buffer will start filling up again at the rate of one element per second.

### 4.5.5 Reading a file lazily

The simplest way to read a file is as a list of characters. However, if the file is very large, this uses an enormous amount of memory. This is why files are usually read incrementally, a block at a time (where a *block* is a contiguous piece of the file). The program is careful to keep in memory only the blocks that are needed. This is memory-efficient, but is cumbersome to program.

Can we have the best of both worlds: to read the file as a list of characters (which keeps programs simple), yet to read in only the parts we need (which saves memory)? With lazy execution the answer is *yes*. Here is the function `ReadListLazy` that solves the problem:

```
fun {ReadListLazy FN}
  {File.readOpen FN}
  fun lazy {ReadNext}
  L T I in
    {File.readBlock I L T}
    if I==0 then T=nil {File.readClose} else T={ReadNext} end
    L
  end
in
  {ReadNext}
end
```

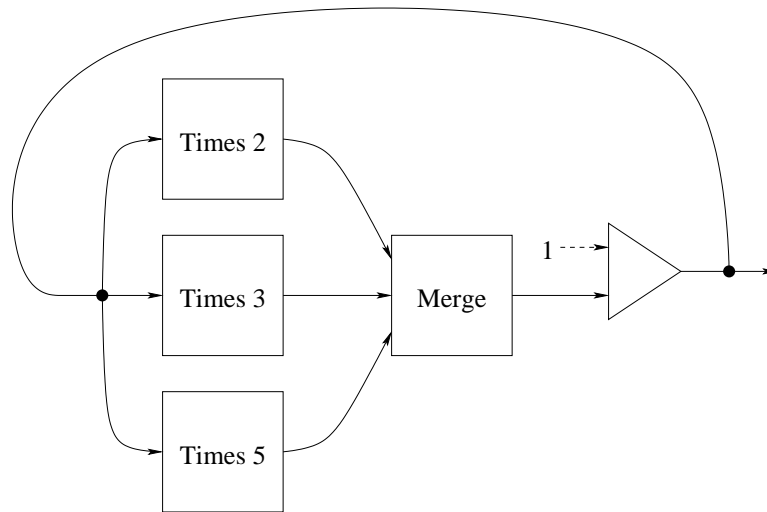


Figure 4.29: Lazy solution to the Hamming problem

It uses three operations in the `File` module (which is available on the book's Web site): `{File.readOpen FN}`, which opens file `FN` for reading, `{File.readBlock I L T}`, which reads a block in the difference list `L#T` and returns its size in `I`, and `{File.readClose}`, which closes the file.

The `ReadListLazy` function reads a file lazily, a block at a time. Whenever a block is exhausted then another block is read automatically. Reading blocks is much more efficient than reading single characters since only one lazy call is needed for a whole block. This means that `ReadListLazy` is practically speaking just as efficient as the solution in which we read blocks explicitly. When the end of file is reached then the tail of the list is bound to `nil` and the file is closed.

The `ReadListLazy` function is acceptable if the program reads *all* of the file, but if it only reads *part* of the file, then it is not good enough. Do you see why not? Think carefully before reading the answer in the footnote!<sup>12</sup> Section 6.9.2 shows the right way to use laziness together with external resources such as files.

### 4.5.6 The Hamming problem

The Hamming problem, named after Richard Hamming, is a classic problem of demand-driven concurrency. The problem is to generate the first  $n$  integers of the form  $2^a 3^b 5^c$  with  $a, b, c \geq 0$ . Hamming actually solved a more general version, which considers products of the first  $k$  primes. We leave this one to an exercise! The idea is to generate the integers in increasing order in a potentially infinite stream. At all times, a finite part  $h$  of this stream is known. To generate the next element of  $h$ , we take the least element  $x$  of  $h$  such that  $2x$  is bigger than the last element of  $h$ . We do the same for 3 and 5, giving  $y$  and  $z$ . Then the next element

<sup>12</sup>It is because the file stays open during the whole execution of the program—this consumes valuable system resources including a file descriptor and a read buffer.

of  $h$  is  $\min(2x, 3y, 5z)$ . We start the process by initializing  $h$  to have the single element 1. Figure 4.29 gives a picture of the algorithm. The simplest way to program this algorithm is with two lazy functions. The first function multiplies all elements of a list by a constant:

```
fun lazy {Times N H}
  case H of X|H2 then N*X|{Times N H2} end
end
```

The second function takes two lists of integers in increasing order and merges them into a single list:

```
fun lazy {Merge Xs Ys}
  case Xs#Ys of (X|Xr)#(Y|Yr) then
    if X<Y then X|{Merge Xr Ys}
    elseif X>Y then Y|{Merge Xs Yr}
    else X|{Merge Xr Yr}
    end
  end
end
```

Each value should appear only once in the output. This means that when  $x==y$ , it is important to skip the value in *both* lists  $x$ s and  $y$ s. With these two functions, it is easy to solve the Hamming problem:

```
H=1|{Merge {Times 2 H}
           {Merge {Times 3 H}
                  {Times 5 H}}}}
{Browse H}
```

This builds a three-argument merge function using two two-argument merge functions. If we execute this as is, then it displays very little:

```
1|_<Future>
```

No elements are calculated. To get the first  $n$  elements of  $H$ , we need to *ask* that they be calculated. For example, we can define the procedure `Touch`:

```
proc {Touch N H}
  if N>0 then {Touch N-1 H.2} else skip end
end
```

This traverses  $N$  elements of  $H$ , which causes them to be calculated. Now we can calculate 20 elements by calling `Touch`:

```
{Touch 20 H}
```

This displays:

```
1|2|3|4|5|6|8|9|10|12|15|16|18|20|24|25|27|30|32|36|_<Future>
```

### 4.5.7 Lazy list operations

All the list functions of Section 3.4 can be made lazy. It is insightful to see how this changes their behavior.

## Lazy append

We start with a simple function: a lazy version of `Append`:

```
fun lazy {LAppend As Bs}
  case As
  of nil then Bs
  [] A|Ar then A|{LAppend Ar Bs}
  end
end
```

The only difference with the eager version is the “lazy” annotation. The lazy definition works because it is recursive: it calculates part of the answer and then calls itself. Calling `LAppend` with two lists will append them lazily:

```
L={LAppend "foo" "bar"}
{Browse L}
```

We say this function is *incremental*: forcing its evaluation only does enough of the calculation to generate one additional output element, and then creates another suspension. If we “touch” successive elements of `L` this will successively show `f`, `o`, `o`, one character at a time. However, after we have exhausted “`foo`”, then `LAppend` is finished, so it will show “`bar`” all at once. How do we make a list append that returns a completely lazy list? One way is to give `LAppend` a lazy list as second argument. First define a function that takes any list and returns a lazy version:

```
fun lazy {MakeLazy Ls}
  case Ls
  of X|Lr then X|{MakeLazy Lr}
  else nil end
end
```

`MakeLazy` works by iterating over its input list, i.e., like `LAppend`, it calculates part of the answer and then calls itself. This only changes the control flow; considered as a function between lists, `MakeLazy` is an identity. Now call `LAppend` as follows:

```
L={LAppend "foo" {MakeLazy "bar"}}
{Browse L}
```

This will lazily enumerate both lists, i.e., it successively returns the characters `f`, `o`, `o`, `b`, `a`, and `r`.

## Lazy mapping

We have seen `Map` in Section 3.6; it evaluates a function on all elements of a list. It is easy to define a lazy version of this function:

```
fun lazy {LMap Xs F}
  case Xs
  of nil then nil
```

```

    [] X|Xr then {F X} | {LMap Xr F}
  end
end

```

This function takes any list or lazy list  $Xs$  and returns a lazy list. Is it incremental?

### Lazy integer lists

We define the function  $\{LFrom\ I\ J\}$  that generates a lazy list of integers from  $I$  to  $J$ :

```

fun {LFrom I J}
  fun lazy {LFromLoop I}
    if I>J then nil else I | {LFromLoop I+1} end
  end
  fun lazy {LFromInf I} I | {LFromInf I+1} end
in
  if J==inf then {LFromInf I} else {LFromLoop I} end
end

```

Why is  $LFrom$  itself not annotated as lazy?<sup>13</sup> This definition allows  $J=inf$ , in which case an infinite lazy stream of integers is generated.

### Lazy flatten

This definition shows that lazy difference lists are as easy to generate as lazy lists. As with the other lazy functions, it suffices to annotate as lazy all recursive functions that calculate part of the solution on each iteration.

```

fun {LFlatten Xs}
  fun lazy {LFlattenD Xs E}
    case Xs
    of nil then E
    [] X|Xr then
      {LFlattenD X {LFlattenD Xr E}}
    [] X then X|E
    end
  end
in
  {LFlattenD Xs nil}
end

```

We remark that this definition has the same asymptotic efficiency as the eager definition, i.e., it takes advantage of the constant-time append property of difference lists.

---

<sup>13</sup>Only recursive functions need to be controlled, since they would otherwise do a potentially unbounded calculation.

### Lazy reverse

Up to now, all the lazy list functions we introduced are incremental, i.e., they are able to produce one element at a time efficiently. Sometimes this is not possible. For some list functions, the work required to produce one element is enough to produce them all. We call these functions *monolithic*. A typical example is list reversal. Here is a lazy definition:

```
fun {LReverse S}
  fun lazy {Rev S R}
    case S
    of nil then R
    [] X|S2 then {Rev S2 X|R} end
  end
in {Rev S nil} end
```

Let us call this function:

```
L={LReverse [a b c]}
{Browse L}
```

What happens if we touch the first element of `L`? This will calculate and display the whole reversed list! Why does this happen? Touching `L` activates the suspension `{Rev [a b c] nil}` (remember that `LReverse` itself is not annotated as lazy). This executes `Rev` and creates a new suspension for `{Rev [b c] [a]}` (the recursive call), but no list pair. Therefore the new suspension is immediately activated. This does another iteration and creates a second suspension, `{Rev [c] [b a]}`. Again, no list pair is available, so the second suspension is immediately activated. This continues until `Rev` returns `[c b a]`. At this point, there is a list pair so the evaluation completes. The need for one list pair has caused the whole list reversal to be done. This is what we mean by a monolithic function. For list reversal, another way to understand this behavior is to think of what list reversal means: the first element of a reversed list is the last element of the input list. We therefore have to traverse the whole input list, which lets us construct the whole reversed list.

### Lazy filter

To complete this section, we give another example of an incremental function, namely filtering an input list according to a condition `F`:

```
fun lazy {LFilter L F}
  case L
  of nil then nil
  [] X|L2 then
    if {F X} then X|{LFilter L2 F} else {LFilter L2 F} end
  end
end
```

We give this function because we will need it for list comprehensions in Section 4.5.9.

### 4.5.8 Persistent queues and algorithm design

In Section 3.4.5 we saw how to build queues with constant-time insert and delete operations. Those queues only work in the *ephemeral* case, i.e., only one version exists at a time. It turns out we can use laziness to build *persistent* queues with the same time bounds. A persistent queue is one that supports multiple versions. We first show how to make an amortized persistent queue with constant-time insert and delete operations. We then show how to achieve worst-case constant-time.

#### Amortized persistent queue

We first tackle the amortized case. The reason why the amortized queue of Section 3.4.5 is not persistent is that `Delete` sometimes does a list reversal, which is not constant time. Each time a `Delete` is done on the same version, another list reversal is done. This breaks the amortized complexity if there are multiple versions.

We can regain the amortized complexity by doing the reverse as part of a lazy function call. Invoking the lazy function creates a suspension instead of doing the reverse right away. Sometime later, when the result of the reverse is needed, the lazy function does the reverse. With some cleverness, this can solve our problem:

- Between the creation of the suspension and the actual execution of the reverse, we arrange that there are enough operations to pay back the costs incurred by the reverse.
- But the reverse can be paid for only once. What if several versions want to do the reverse? This is not a problem. Laziness guarantees that the reverse is only done once, even if more than one version triggers it. The first version that needs it will activate the trigger and save the result. Subsequent versions will use the result without doing any calculation.

This sounds nice, but it depends on being able to create the suspension far enough in advance of the actual reverse. Can we do it? In the case of a queue, we can. Let us represent the queue as a 4-tuple:

`q(LenF F LenR R)`

`F` and `R` are the front and rear lists, like in the ephemeral case. We add the integers `LenF` and `LenR`, which give the lengths of `F` and `R`. We need these integers to test when it is time to create the suspension. At some magic instant, we move the elements of `R` to `F`. The queue then becomes:

`q(LenF+LenR {LAppend F {Reverse R}} 0 nil)`



In Section 3.4.5 we did this (eagerly) when  $F$  became empty, so the `Append` did not take any time. But this is too late to keep the amortized complexity, since the reverse is not paid for (e.g., maybe  $R$  is a very big list). We remark that the reverse gets evaluated in any case when the `LAppend` has finished, i.e., after  $|F|$  elements are removed from the queue. Can we arrange that the elements of  $F$  pay for the reverse? We can, if we create the suspension when  $|R| \approx |F|$ . Then removing each element of  $F$  pays for part of the reverse. By the time we have to evaluate the reverse, it is completely paid for. Using the lazy append makes the payment incremental. This gives the following implementation:

```

fun {NewQueue} q(0 nil 0 nil) end

fun {Check Q}
  case Q of q(LenF F LenR R) then
    if LenF >= LenR then Q
    else q(LenF+LenR {LAppend F {Reverse R}} 0 nil) end
  end
end

fun {Insert Q X}
  case Q of q(LenF F LenR R) then
    {Check q(LenF F LenR+1 X|R)}
  end
end

fun {Delete Q X}
  case Q of q(LenF F LenR R) then F1 in
    F=X|F1 {Check q(LenF-1 F1 LenR R)}
  end
end

```

Both `Insert` and `Delete` call the function `Check`, which chooses the moment to do the lazy call. Since `Insert` increases  $|R|$  and `Delete` decreases  $|F|$ , eventually  $|R|$  becomes as large as  $|F|$ . When  $|R| = |F| + 1$ , `Check` does the lazy call `{LAppend F {Reverse R}}`. The function `LAppend` is defined in Section 4.5.7.

Let us summarize this technique. We replace the original eager function call by a lazy function call. The lazy call is partly incremental and partly monolithic. The trick is that the lazy call starts off being incremental. By the time the monolithic part is reached, there have been enough incremental steps so that the monolithic part is paid for. It follows that the result is amortized constant-time.

For a deeper discussion of this technique including its application to other data structures and a proof of correctness, we recommend [138].

### Worst-case persistent queue

The reason the above definition is not worst-case constant-time is because `Reverse` is monolithic. If we could rewrite it to be incremental, then we would have a so-

lution with constant-time worst-case behavior. But list reversal cannot be made incremental, so this does not work. Let us try another approach.

Let us look at the context of the call to `Reverse`. It is called together with a lazy append:

```
{LAppend F {Reverse R}}
```

This first executes the append incrementally. When all elements of `F` have been passed to the output, then the reverse is executed monolithically. The cost of the reverse is amortized over the steps of the append.

Instead of amortizing the cost of the reverse, perhaps we can actually *do* the reverse together with the steps of the append. When the append is finished, the reverse will be finished as well. This is the heart of the solution. To implement it, let us compare the definitions of reverse and append. Reverse uses the recursive function `Rev`:

```
fun {Reverse R}
  fun {Rev R A}
    case R
    of nil then A
    [] X|R2 then {Rev R2 X|A} end
  end
in {Rev R nil} end
```

`Rev` traverses `R`, accumulates a solution in `A`, and then returns the solution. Can we do both `Rev` and `LAppend` in a single loop? Here is `LAppend`:

```
fun lazy {LAppend F B}
  case F
  of nil then B
  [] X|F2 then X|{LAppend F2 B}
  end
end
```

This traverses `F` and returns `B`. The recursive call is passed `B` unchanged. Let us change this to use `B` to accumulate the result of the reverse! This gives the following combined function:

```
fun lazy {LAppRev F R B}
  case F#R
  of nil#[Y] then Y|B
  [] (X|F2)#(Y|R2) then X|{LAppRev F2 R2 Y|B}
  end
end
```

`LAppRev` traverses both `F` and `R`. During each iteration, it calculates one element of the append and accumulates one element of the reverse. This definition only works if `R` has *exactly one more element* than `F`, which is true for our queue. The original call:

```
{LAppend F {Reverse R}}
```

is replaced by:

```
{LAppRev F R nil}
```

which gives exactly the same result except that `LAppRev` is completely incremental. The definition of `Check` then becomes:

```
fun {Check Q}
  case Q of q(LenF F LenR R) then
    if LenR=<LenF then Q
    else q(LenF+LenR {LAppRev F R nil} 0 nil) end
  end
end
```

Careful analysis shows that the worst-case bound of this queue is  $O(\log n)$ , and not  $O(1)$  as our intuition might expect it to be. The bound is much better than  $O(n)$ , but it is not constant. See the Exercises for an explanation and a suggestion on how to achieve a constant bound.

Taking a program with a worst-case bound and adding laziness naively will give an amortized bound. This is because laziness changes where the function calls are executed, but does not do more of them (the eager case is an upper bound). The definition of this section is remarkable because it does just the opposite: it starts with an amortized bound and uses laziness to give a worst-case bound.

### Lessons for algorithm design

Laziness is able to shuffle calculations around, spreading them out or bunching them together without changing the final result. This is a powerful tool for designing declarative algorithms. It has to be used carefully, however. Used naively, laziness can destroy perfectly good worst-case bounds, turning them into amortized bounds. Used wisely, laziness can improve amortized algorithms: it can sometimes make the algorithm persistent and it can sometimes transform the amortized bound into a worst-case bound.

We can outline a general scheme. Start with an algorithm `A` that has an amortized bound  $O(f(n))$  when used ephemerally. For example, the first queue of Section 3.4.5 has an amortized bound of  $O(1)$ . We can use laziness to move from ephemeral to persistent while keeping this time bound. There are two possibilities:

- Often we can get a modified algorithm `A'` that keeps the amortized bound  $O(f(n))$  when used persistently. This is possible when the expensive operations can be spread out to be mostly incremental but with a few remaining monolithic operations.
- In a few cases, we can go farther and get a modified algorithm `A''` with *worst-case* bound  $O(f(n))$  when used persistently. This is possible when the expensive operations can be spread out to be completely incremental.

This section realizes both possibilities with the first queue of Section 3.4.5. The persistent algorithms so obtained are often quite efficient, especially if used by applications that really need the persistence. They compare favorably with algorithms in stateful models.

### 4.5.9 List comprehensions

List comprehensions are a powerful tool for calculating with lazy streams. They allow to specify lazy streams in a way that closely resembles the mathematical notation of set comprehension. For example, the mathematical notation  $\{x * y \mid 1 \leq x \leq 10, 1 \leq y \leq x\}$  specifies the set  $\{1*1, 2*1, 2*2, 3*1, 3*2, 3*3, \dots, 10*10\}$ , i.e.  $\{1, 2, 3, 4, 5, \dots, 100\}$ . We turn this notation into a practical programming tool by modifying it to specify not *sets*, but *lazy streams*. This makes the notation very efficient to implement, while keeping it at a high level of abstraction. For example, the list comprehension  $[x * y \mid 1 \leq x \leq 10, 1 \leq y \leq x]$  (notice the square list brackets!) specifies the list  $[1*1 \ 2*1 \ 2*2 \ 3*1 \ 3*2 \ 3*3 \ \dots \ 10*10]$  (in this order), i.e., the list  $[1 \ 2 \ 4 \ 3 \ 6 \ 9 \ \dots \ 100]$ . The list is calculated lazily. Because of laziness the list comprehension can generate a potentially unbounded stream, not just a finite list.

List comprehensions have the following basic form:

$$[f(x) \mid x \leftarrow \text{generator}(a_1, \dots, a_n), \text{guard}(x, a_1, \dots, a_n)]$$

The generator  $x \leftarrow \text{generator}(a_1, \dots, a_n)$  calculates a lazy list whose elements are successively assigned to  $x$ . The guard  $\text{guard}(x, a_1, \dots, a_n)$  is a boolean function. The list comprehension specifies a lazy list containing the elements  $f(x)$ , where  $f$  is any function and  $x$  takes on values from the generator for which the guard is true. In the general case, there can be any number of variables, generators, and guards. A typical generator is *from*:

$$x \leftarrow \text{from}(a, b)$$

Here  $x$  takes on the integer values  $a, a+1, \dots, b$ , in that order. Calculation is done from left to right. The generators, when taken from left to right, are considered as nested loops: the rightmost generator is the innermost loop.

There is a close connection between list comprehensions and the relational programming of Section 9. Both provide lazy interfaces to infinitely long sequences and make it easy to write “generate-and-test” programs. Both allow to specify the sequences in a declarative way.

While list comprehensions are usually considered to be lazy, they can in fact be programmed in both eager and lazy versions. For example, the list comprehension:

$$z = [x \# x \mid x \leftarrow \text{from}(1, 10)]$$

can be programmed in two ways. An eager version is:

```
Z = {Map {From 1 10} fun {$ X} X#X end}
```

For the eager version, the declarative model of Chapter 2 is good enough. It uses the `Map` function of Section 3.6.3 and the `From` function which generates a list of integers. A lazy version is:

```
Z = {LMap {LFrom 1 10} fun {$ X} X#X end}
```

The lazy version uses the `LMap` and `LFrom` functions of the previous section. This example and most examples of this section can be done with either a lazy or eager version. Using the lazy version is always correct. Using the eager version is a performance optimization. It is several times faster if the cost of calculating the list elements is not counted. The optimization is only possible if the whole list fits in memory. In the rest of this section, we always use the lazy version.

Here is a list comprehension with two variables:

$$z = [x\#y \mid x \leftarrow \text{from}(1, 10), y \leftarrow \text{from}(1, x)]$$

This can be programmed as:

```
Z = {LFlatten
  {LMap {LFrom 1 10} fun {$ X}
    {LMap {LFrom 1 X} fun {$ Y}
      X#Y
    end}
  end}}
```

We have seen `LFlatten` in the previous section; it converts a list of lists to a “flat” lazy list, i.e., a lazy list that contains all the elements, but no lists. We need `LFlatten` because otherwise we have a list of lists. We can put `LFlatten` inside `LMap`:

```
fun {FMap L F}
  {LFlatten {LMap L F}}
end
```

This simplifies the program:

```
Z = {FMap {LFrom 1 10} fun {$ X}
  {LMap {LFrom 1 X} fun {$ Y}
    X#Y
  end}
end}
```

Here is an example with two variables and a guard:

$$z = [x\#y \mid x \leftarrow \text{from}(1, 10), y \leftarrow \text{from}(1, 10), x + y \leq 10]$$

This gives the list of all pairs  $x\#y$  such that the sum  $x + y$  is at most 10. It can be programmed as:

```
Z = {LFilter
  {FMap {LFrom 1 10} fun {$ X}
```

```

      {LMap {LFrom 1 10} fun {$ Y}
        X#Y
      end}
    end}
  fun {$ X#Y} X+Y=<10 end}

```

This uses the function `LFilter` defined in the previous section. We can reformulate this example to be more efficient. The idea is to generate as few elements as possible. In the above example, 100 ( $=10*10$ ) elements are generated. From  $2 \leq x + y \leq 10$  and  $1 \leq y \leq 10$ , we derive that  $1 \leq y \leq 10 - x$ . This gives the following solution:

$$z = [x\#y \mid x \leftarrow \text{from}(1, 10), y \leftarrow \text{from}(1, 10 - x)]$$

The program then becomes:

```

Z = {FMap {LFrom 1 10} fun {$ X}
      {LMap {LFrom 1 10-X} fun {$ Y}
        X#Y
      end}
    end}

```

This gives the same list as before, but only generates about half as many elements.

## 4.6 Soft real-time programming

### 4.6.1 Basic operations

The `Time` module contains a number of useful soft real-time operations. A *real-time* operation has a set of deadlines (particular times) at which certain calculations must be completed. A *soft real-time* operation requires only that the real-time deadlines be respected most of the time. This is opposed to *hard real-time*, which has hard deadlines, i.e., that must be respected *all* the time, without any exception. Hard real-time is needed when lives are at stake, e.g., in medical equipment and air traffic control. Soft real-time is used in other cases, e.g., for telephony and consumer electronics. Hard real-time requires special techniques for both hardware and software. Standard personal computers cannot do hard real-time because they have unpredictable hardware delays (e.g., virtual memory, caching, process scheduling). Soft real-time is much easier to implement and is often sufficient. Three soft real-time operations provided by `Time` are:

- `{Delay I}`: suspends the executing thread for at least `I` milliseconds and then continues.
- `{Alarm I U}`: creates a new thread that binds `U` to `unit` after at least `I` milliseconds. `Alarm` can be implemented with `Delay`.
- `{Time.time}`: returns the integer number of seconds that have passed since the current year started.

```

local
  proc {Ping N}
    if N==0 then {Browse 'ping terminated'}
    else {Delay 500} {Browse ping} {Ping N-1} end
  end
  proc {Pong N}
    {For 1 N 1
      proc {$ I} {Delay 600} {Browse pong} end}
    {Browse 'pong terminated'}
  end
in
  {Browse 'game started'}
  thread {Ping 50} end
  thread {Pong 50} end
end

```

Figure 4.30: A simple ‘Ping Pong’ program

The semantics of `Delay` is simple: it communicates to the scheduler that the thread is to be considered suspended for a given time period. After this time is up, the scheduler marks the thread as runnable again. The thread is not necessarily run immediately. If there are lots of other runnable threads, it may take some time before the thread actually runs.

We illustrate the use of `Delay` by means of a simple example that shows the interleaving execution of two threads. The program is called ‘Ping Pong’ and is defined in Figure 4.30. It starts two threads. One displays `ping` periodically each 500 milliseconds and the other displays `pong` each 600 milliseconds. Because pongs come out slower than pings, it is possible for two pings to be displayed without any pongs in between. Can the same thing happen with two pongs? That is, can two pongs ever be displayed with no pings in between? Assume that the `Ping` thread has not yet terminated, otherwise the question would be too easy. Think carefully before reading the answer in the footnote.<sup>14</sup>

### A simple standalone application

Section 3.9 in Chapter 2 shows how to make standalone applications in Oz. To make the ‘Ping Pong’ program standalone, the first step is to make a functor of it, as shown in Figure 4.31. If the source code is stored in file `PingPong.oz`, then the program can be compiled with the following command:

```
ozc -x PingPong.oz
```

---

<sup>14</sup>The language does indeed allow two pongs to be displayed with no intervening pings because the definition of `Delay` only gives the *minimum* suspension time. The thread suspending for 500 milliseconds can occasionally suspend for a longer time, for example for 700 milliseconds. But this is a rare occurrence in practice because it depends on external events in the operating system or in other threads.

```

functor
import
    Browser(browse:Browse)
define
    proc {Ping N}
        if N==0 then {Browse `ping terminated`}
        else {Delay 500} {Browse ping} {Ping N-1} end
    end
    proc {Pong N}
        {For 1 N 1
            proc {$ I} {Delay 600} {Browse pong} end }
        {Browse `pong terminated`}
    end
in
    {Browse `game started`}
    thread {Ping 50} end
    thread {Pong 50} end
end

```

Figure 4.31: A standalone ‘Ping Pong’ program

Type `PingPong` in your shell to start the program. To terminate this program in a Unix shell you have to type CTRL-C.

The program of Figure 4.31 does not terminate properly when the `Ping` and the `Pong` threads terminate. It does not detect when the threads terminate. We can fix this problem using the techniques of Section 4.4.3. Figure 4.32 adds a termination detection that terminates the main thread only when both the `Ping` and the `Pong` threads terminate. We could also use the `Barrier` abstraction directly. After detecting termination, we use the call `{Application.exit 0}` to cleanly exit the application.

## 4.6.2 Ticking

We would like to invoke an action (e.g., send a message to a stream object, call a procedure, etc.) *exactly* once per second, giving it the local time as argument. We have three operations at our disposal: `{Delay D}`, which delays for at least `D` milliseconds, `{Time.time}`, which returns the number of seconds since January 1 of the current year, and `{OS.localTime}`, which returns a record giving local time accurate to one second. How does the following function measure up:

```

fun {NewTicker}
    fun {Loop}
        X={OS.localTime}
    in
        {Delay 1000}
        X|{Loop}

```



```

functor
import
    Browser(browse:Browse)
    Application
define
    ...
    X1 X2
in
    {Browse `game started`}
    thread {Ping 50} X1=unit end
    thread {Pong 50} X2=unit end
    {Wait X1} {Wait X2}
    {Application.exit 0}
end

```

Figure 4.32: A standalone ‘Ping Pong’ program that exits cleanly

```

    end
in
    thread {Loop} end
end

```

This function creates a stream that grows by one element per second. To execute an action once every second, create a thread that reads the stream and performs the action:

```

thread for X in {NewTicker} do {Browse X} end end

```

Any number of threads can read the same stream. The problem is, this solution is not quite right. The stream is extended *almost exactly* once per second. The problem is the “almost”. Every once in a while, one second is lost, i.e., successive elements on the stream show a difference of two seconds. However, there is one good point: the same second cannot be sent twice, since {Delay 1000} guarantees a delay of at least 1000 milliseconds, to which is added the execution of the instructions in Loop. This gives a total delay of at least  $1000 + \epsilon$  milliseconds, where  $\epsilon$  is a fraction of a microsecond.

How can we correct this problem? A simple way is to compare the current result of `OS.localTime` with the previous result, and to add an element to the stream only when the local time changes. This gives:

```

fun {NewTicker}
    fun {Loop T}
        T1={OS.localTime}
    in
        {Delay 900}
        if T1\=T then T1|{Loop T1} else {Loop T1} end
    end
in

```

```

    thread {Loop {OS.localTime}} end
end

```

This version guarantees that *exactly* one tick will be sent per second, if {Delay 900} always delays for less than one second. The latter condition holds if there are not too many active threads *and* garbage collection does not take too long. One way to *guarantee* the first condition is to give the Loop thread high priority and all other threads medium or low priority. To guarantee the second condition, the program must ensure that there is not too much active data, since garbage collection time is proportional to the amount of active data.

This version has the minor problem that it “hesitates” every 9 seconds. That is, it can happen that {OS.localTime} gives the same result twice in a row, since the two calls are separated by just slightly more than 900 milliseconds. This means that the stream will not be updated for 1800 milliseconds. Another way to see this problem is that 10 intervals of 900 milliseconds are needed to cover 9 seconds, which means that nothing happens during one of the intervals. How can we avoid this hesitation? A simple way is to make the delay smaller. With a delay of 100 milliseconds, the hesitation will never be greater than 100 milliseconds plus the garbage collection time.

A better way to avoid the hesitation is to use *synchronized clocks*. That is, we create a free-running counter that runs at approximately one second per tick, and we adjust its speed so that it remains synchronized with the operating system time. Here is how it is done:

```

fun {NewTicker}
  fun {Loop N}
    T={Time.time}
  in
    if T>N then {Delay 900}
    elseif T<N then {Delay 1100}
    else {Delay 1000} end
    N|{Loop N+1}
  end
in
  thread {Loop {Time.time}} end
end

```

The loop has a counter, N, that is always incremented by one. We compare the counter value to the result of {Time.time}.<sup>15</sup> If the counter is slower (T>N), we speed it up. Likewise, if the counter is faster (T<N), we slow it down. The speedup and slowdown factors are small (10% in the example), which makes the hesitation unnoticeable.

---

<sup>15</sup>How would you fix NewTicker to work correctly when Time.time turns over, i.e., goes back to 0?

## 4.7 Limitations and extensions of declarative programming

Declarative programming has the major advantage that it considerably simplifies system building. Declarative components can be built and debugged independently of each other. The complexity of a system is the *sum* of the complexities of its components. A natural question to ask is how far can declarative programming go? Can everything be programmed in a declarative way, such that programs are both natural and efficient? This would be a major boon for system building. We say a program is *efficient* if its performance differs by just a constant factor from the performance of an assembly language program to solve the same problem. We say a program is *natural* if very little code is needed just for technical reasons unrelated to the problem at hand. Let us consider efficiency and naturalness issues separately. There are three naturalness issues: modularity, nondeterminism, and interfacing with the real world.

We recommend to use the declarative model of this chapter or the sequential version of Chapter 2 except when any of the above issues is critical. This makes it easier to write correct and efficient components.

### 4.7.1 Efficiency

Is declarative programming efficient? There is a fundamental mismatch between the declarative model and a standard computer, such as presented in [146]. The computer is optimized for modifying data in-place, while the declarative model never modifies data but always creates new data. This is not as severe a problem as it seems at first glance. The declarative model may have a large inherent memory consumption, but its active memory size remains small. The task remains, though, to implement the declarative model with in-place assignment. This depends first on the sophistication of the compiler.

Can a compiler map declarative programs effectively to a standard computer? Paraphrasing science fiction author and futurologist Arthur C. Clarke, we can say that “any sufficiently advanced compiler is indistinguishable from magic” [37].<sup>16</sup> That is, it is unrealistic to expect the compiler to rewrite your program. Even after several decades of research, no such compiler exists for general-purpose programming. The farthest we have come is compilers that can rewrite the program in particular cases. Computer scientist Paul Hudak calls them “smart-aleck” compilers. Because of their unpredictable optimizations, they are hard to use. Therefore, for the rest of the discussion, we assume that the compiler does a straightforward mapping from the source program to the target, in the sense that time and space complexities of compiled code can be derived in a simple way from language semantics.

---

<sup>16</sup>Clarke’s Third Law: “Any sufficiently advanced technology is indistinguishable from magic.”

Now we can answer the question whether declarative programming is efficient. Given a straightforward compiler, the pedantic answer to the question is *no*. But in fact the practical answer is *yes*, with one caveat: declarative programming is efficient *if* one is allowed to rewrite the program to be less natural. Here are three typical examples:

1. A program that does *incremental modifications* of large data structures, e.g., a simulation that modifies large graphs (see Section 6.8.4), cannot in general be compiled efficiently. Even after decades of research, there is no straightforward compiler that can take such a program and implement it efficiently. However, if one is allowed to rewrite the program, then there is a simple trick that is often sufficient in practice. If the state is threaded (e.g., kept in an accumulator) and the program is careful never to access an old state, then the accumulator can be implemented with destructive assignment.
2. A function that does *memoization* cannot be programmed without changing its interface. Assume we have a function that uses many computational resources. To improve its performance, we would like to add *memoization* to it, i.e., an internal cache of previously-calculated results, indexed by the function arguments. At each function call, we first check the cache to see if the result is already there. This internal cache cannot be added without rewriting the program by threading an accumulator everywhere that the function is called. Section 10.3.2 gives an example.
3. A function that *implements a complex algorithm* often needs intricate code. That is, even though the program *can* be written declaratively with the same efficiency as a stateful program, doing so makes it more complex. This follows because the declarative model is less expressive than the stateful model. Section 6.8.1 shows an example: a transitive closure algorithm written in both the declarative and the stateful models. Both versions have time efficiency  $O(n^3)$ . The stateful algorithm is simpler to write than the declarative one.

We conclude that declarative programming cannot always be efficient and natural simultaneously. Let us now look at the naturalness issues.

### 4.7.2 Modularity

We say a program is *modular* with respect to a change in a given part if the change can be done without changing the rest of the program. Modularity is discussed further in Section 6.7.2. Here are two examples where declarative programs are not modular:

1. The first example is the memoization cache we saw before. Adding this cache to a function is not modular, since an accumulator must be threaded in many places outside the function.

2. A second example is instrumenting a program. We would like to know how many times some of its subcomponents are invoked. We would like to add counters to these subcomponents, preferably without changing either the subcomponent interfaces or the rest of the program. If program is declarative, this is impossible, since the only way is to thread an accumulator throughout the program.

Let us look closer at the second example. Assume that we are using the declarative model to implement a large declarative component. The component definition looks something like this:

```

fun {SC ...}
  proc {P1 ...}
    ...
  end
  proc {P2 ...}
    ...
    {P1 ...}
    {P2 ...}
  end
  proc {P3 ...}
    ...
    {P2 ...}
    {P3 ...}
  end
in
  `export` (p1:P1 p2:P2 p3:P3)
end

```

Calling SC instantiates the component: it returns a module with three operations, P1, P2, and P3. We would like to instrument the component by counting the number of times procedure P1 is called. The successive values of the count are a state. We can encode this state as an accumulator, i.e., by adding two arguments to each procedure. With this added instrumentation, the component definition looks something like this:

```

fun {SC ...}
  proc {P1 ... S1 ?Sn}
    Sn=S1+1
    ...
  end
  proc {P2 ... T1 ?Tn}
    ...
    {P1 ... T1 T2}
    {P2 ... T2 Tn}
  end
  proc {P3 ... U1 ?Un}
    ...

```

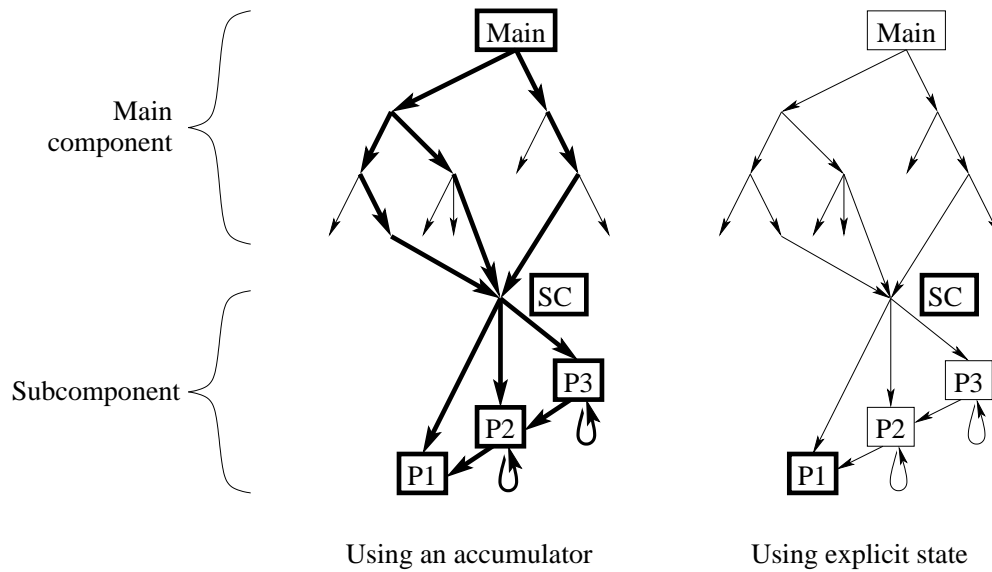


Figure 4.33: Changes needed for instrumenting procedure P1

```

    {P2 ... U1 U2}
    {P3 ... U2 Un}
  end
in
  ^export^(p1:P1 p2:P2 p3:P3)
end

```

Each procedure defined by SC has a changed interface: it has two extra arguments that together form an accumulator. The procedure P1 is called as {P1 ... Sin Sout}, where Sin is the input count and Sout is the output count. The accumulator has to be threaded between the procedure calls. This technique requires both SC and the calling module to do a fair amount of bookkeeping, but it works.

Another solution is to write the component in a stateful model. One such model is defined in Chapter 6; for now assume that we have a new language entity, called “cell”, that we can assign and access (with the := and @ operators), similar to an assignable variable in imperative programming languages. Cells were introduced in Chapter 1. Then the component definition looks something like this:

```

fun {SC ...}
  Ctr={NewCell 0}
  proc {P1 ...}
    Ctr:=@Ctr+1
    ...
  end
  proc {P2 ...}
    ...

```