```
      {P1 ...}
      {P2 ...}
   end
   proc {P3 ...}
      ...
      {P2 ...}
      {P3 ...}
   end
   fun {Count} @Ctr end
in
   ´export´(p1:P1 p2:P2 p3:P3 count:Count)
end
```

In this case, the component interface has one extra function, `Count`, and the interfaces to `P1`, `P2`, and `P3` are unchanged. The calling module has no bookkeeping to do whatsoever. The count is automatically initialized to zero when the component is instantiated. The calling module can call `Count` at any time to get the current value of the count. The calling module can also ignore `Count` completely, if it likes, in which case the component has exactly the same behavior as before (except for a very slight difference in performance).

Figure 4.33 compares the two approaches. The figure shows the call graph of a program with a component `Main` that calls subcomponent `SC`. A *call graph* is a directed graph where each node represents a procedure and there is an edge from each procedure to the procedures it calls. In Figure 4.33, `SC` is called from three places in the main component. Now let us instrument `SC`. In the declarative approach (at left), an accumulator has to be added to each procedure on the path from `Main` to `P1`. In the stateful approach (at right), the only changes are the extra operation `Count` and the body of `P1`. In both cases, the changes are shown with thick lines. Let us compare the two approaches:

- The declarative approach is not modular with respect to instrumenting `P1`, because *every* procedure definition and call on the path from `Main` to `P1` needs two extra arguments. The interfaces to `P1`, `P2`, and `P3` are all changed. This means that other components calling `SC` have to be changed too.

- The stateful approach is modular because the cell is mentioned only where it is needed, in the initialization of `SC` and in `P1`. In particular, the interfaces to `P1`, `P2`, and `P3`, remain the same in the stateful approach. Since the extra operation `Count` can be ignored, other components calling `SC` do not have to be changed.

- The declarative approach is slower because it does much extra argument passing. All procedures are slowed down for the sake of one. The stateful approach is efficient; it only spends time when necessary.

Which approach is simpler: the first or the second? The first has a simpler model but a more complex program. The second has a more complex model but a

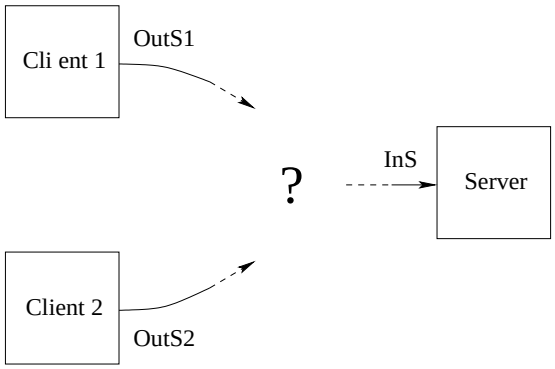

Figure 4.34: How can two clients send to the same server? They cannot!

simpler program. In our view, the declarative approach is not natural. Because it is modular, the stateful approach is clearly the simplest overall.

The fallacy of the preprocessor

Maybe there is a way we can have our cake and eat it too. Let us define a preprocessor to add the arguments so we do not have to write them everywhere. A *preprocessor* is a program that takes another program's source code as input, transforms it according to some simple rules, and returns the result. We define a preprocessor that takes the syntax of the stateful approach as input and translates it into a program that looks like the declarative approach. Voilà! It seems that we can now program with state in the declarative model. We have overcome a limitation of the declarative model. But have we? In fact, we have done nothing of the sort. All we have succeeded in doing is build an inefficient implementation of a stateful model. Let us see why:

- When using the preprocessor, we see only programs that look like the stateful version, i.e., stateful programs. This obliges us to reason in the stateful model. We have therefore *de facto* extended the declarative model with explicit state.
- The preprocessor transforms these stateful programs into programs with threaded state, which are inefficient because of all the argument passing.

4.7.3 Nondeterminism

The declarative concurrent model seems to be quite powerful for building concurrent programs. For example, we can easily build a simulator for digital electronic circuits. However, despite this apparent power, the model has a limitation that cripples it for many concurrent applications: it always behaves deterministically. If a program has observable nondeterminism, then it is not declarative. This limitation is closely related to modularity: components that are truly independent

behave nondeterministically with respect to each other. To show that this is not a purely theoretical limitation, we give two realistic examples: a client/server application and a video display application.

The limitation can be removed by adding a nondeterministic operation to the model. The extended model is no longer declarative. There are many possible nondeterministic operations we could add. Chapters 5 and 8 explain the possibilities in detail. Let us briefly go over them here:

- A first solution is to add a nondeterministic wait operation, such as `WaitTwo` which waits for one of two variables to become bound, and indicates one of the bound ones. Its definition is given in the supplements file on the book's Web site. `WaitTwo` is nice for the client/server application.
- A second solution is to add `IsDet`, a boolean function that tests immediately whether a dataflow variable is bound or not. This allows to use dataflow variables as a weak form of state. `IsDet` is nice for the video display application.
- A third solution is to add explicit state to the model, for example in the form of ports (communication channels) or cells (mutable variables).

How do these three solutions compare in expressiveness? `WaitTwo` can be programmed in the declarative concurrent model with explicit state. Therefore, it seems that the most expressive model needs just explicit state and `IsDet`.

A client/server application

Let us investigate a simple client/server application. Assume that there are two independent clients. Being independent implies that they are concurrent. What happens if they communicate with the same server? Because they are independent, the server can receive information in any order from the two clients. This is observable nondeterministic behavior.

Let us examine this closer and see why it cannot be expressed in the declarative concurrent model. The server has an input stream from which it reads commands. Let us start with one client, which sends commands to the server. This works perfectly. How can a second client connect to the server? The second client has to obtain a reference to a stream that it can bind and that is read by the server. The problem is that such a stream does not exist! There is only one stream, between the first client and the server. The second client cannot bind that stream, since this would conflict with the first client's bindings.

How can we solve this problem? Let us approach it naively and see if we can find a solution. One approach might be to let the server have two input streams, like this:

```
fun {Server InS1 InS2}
   ...
end
```

But how does the server read the streams? Does it first read one element from `InS1` and then one element from `InS2`? Does it simultaneously read one element from both streams? Neither of these solutions is correct. In fact, it is not possible to write a solution in the declarative concurrent model. The only thing we can do is have two independent servers, one for each client. But these servers cannot communicate with each other, since otherwise we would have the same problem all over again.

Figure 4.34 illustrates the problem: `InS` is the server's input stream and `OutS1` and `OutS2` are the two client's output streams. How can the messages appearing on both client streams be given to the server? The simple answer is that in the declarative concurrent model they cannot! In the declarative concurrent model, an active object always has to know from which stream it will read next.

How can we solve this problem? If the clients execute in coordinated fashion, so that the server always knows which client will send the next command, then the program is declarative. But this is unrealistic. To write a true solution, we have to add a nondeterministic operation to the model, like the `WaitTwo` operation we mentioned above. With `WaitTwo`, the server can wait for a command from either client. Chapter 5 gives a solution using `WaitTwo`, in the nondeterministic concurrent model (see Advanced Topics).

A video display application

Let us look at a simple video display application. It consists of a displayer that receives a stream of video frames and displays them. The frames arrive at a particular rate, that is, some number of frames arrive per second. For various reasons, this rate can fluctuate: the frames have different resolutions, some processing might be done on them, or the transmission network has varying bandwidth and latency.

Because of the varying arrival rate, the displayer cannot always display all frames. Sometimes it has to *skip over* frames. For example, it might want to skip quickly to the latest frame that was sent. This kind of stream management cannot be done in the declarative concurrent model, because there is no way to detect the end of the stream. It can be done by extending the model with one new operation, `IsDet`. The boolean test `{IsDet Xs}` checks immediately whether `Xs` is already bound or not (returning **`true`** or **`false`**), and does not wait if it is not bound. Using `IsDet`, we can define the function `Skip` that takes a stream and returns its unbound tail:

```
fun {Skip Xs}
   if {IsDet Xs} then
      case Xs of _|Xr then {Skip Xr} [] nil then nil end
   else Xs end
end
```

This iterates down the stream until it finds an unbound tail. Here is a slightly different version that always waits until there is at least one element:

```
fun {Skip1 Xs}
   case Xs of X|Xr then
      if {IsDet Xr} then {Skip1 Xr} else Xs end
   [] nil then nil end
end
```

With `Skip1`, we can write a video displayer that, after it has displayed a frame, immediately skips to the latest transmitted frame:

```
proc {Display Xs}
   case {Skip1 Xs}
   of X|Xr then
      {DisplayFrame X}
      {Display Xr}
   [] nil then skip
   end
end
```

This will work well even if there are variations in the frame arrival rate and the time to display a frame.

4.7.4 The real world

The real world is not declarative. It has both state (entities have an internal memory) and concurrency (entities evolve independently).[17] Since declarative programs interact with the real world, either directly or indirectly, they are part of an environment that contains these concepts. This has two consequences:

1. *Interfacing problems.* Declarative components lack the expressivity to interface with non-declarative components. The latter are omnipresent, e.g., hardware peripherals and user interfaces are both inherently concurrent and stateful (see Section 3.8). Operating systems also use concurrency and state for their own purposes, because of the reasons mentioned previously. One might think that these non-declarative properties could be either masked or encoded somehow, but somehow this never works. Reality always peeks through.

2. *Specification problems.* Program specifications often mention state and concurrency, because they are targeted for the real world. If the program is declarative, then it has to encode this in some way. For example, a specification for a collaborative tool may require that each user lock what they are working on to prevent conflicts during concurrent access. In the implementation, the locks have to be encoded in some way. Using locks directly in a stateful model gives an implementation that is closer to the specification.

[17] In fact, the real world is parallel, but this is modeled inside a program with concurrency. *Concurrency* is a language concept that expresses logically independent computations. *Parallelism* is an implementation concept that expresses activities that happen simultaneously. In a computer, parallelism is used only to increase performance.

4.7.5 Picking the right model

There exist many computation models that differ in how expressive they are and how hard it is to reason about programs written in them. The declarative model is one of the simplest of all. However, as we have explained, it has serious limitations for some applications. There are more expressive models that overcome these limitations, at the price of sometimes making reasoning more complicated. For example, concurrency is often needed when interacting with the external world. When such interactions are important then a concurrent model should be used instead of trying to get by with just the declarative model.

The more expressive models are not "better" than the others, since they do not always give simpler programs and reasoning in them is usually harder.[18] In our experience, all models have their place and can be used together to good effect in the same program. For example, in a program with concurrent state, many components can be declarative. Conversely, in a declarative program, some components (e.g., graph algorithms) need state to be implemented well. We summarize this experience in the following rule:

> **Rule of least expressiveness**
>
> When programming a component, the right computation model for the component is the least expressive model that results in a natural program.

The idea is that each component should be programmed in its "natural" model. Using a less expressive model would give a more complex program and using a more expressive model would not give a simpler program but would make reasoning about it harder.

The problem with this rule is that we have not really defined "natural". This is because to some degree, naturalness is a subjective property. Different people may find different models easier to use, because of their differing knowledge and background. The issue is not the precise definition of "natural", but the fact that such a definition *exists* for each person, even though it might be different for different people.

4.7.6 Extended models

Now we have some idea of the limitations of the declarative model and a few intuitions on how extended models with state and concurrency can overcome these limitations. Let us therefore give a brief overview of the declarative model and its extensions:

- **Declarative sequential model** (see Chapters 2–3). This model encompasses strict functional programming and deterministic logic programming.

[18] Another reason why they are not better has to do with distributed programming and network-awareness, which is explained in Chapter 11.

It extends the former with partial values (using dataflow variables, which are also called "logic variables") and the latter with higher-order procedures. Reasoning with this model is based on algebraic calculations with values. Equals can be substituted for equals and algebraic identities can be applied. A component's behavior is independent of when it is executed or of what happens in the rest of the computation.

- **Declarative concurrent model** (in this chapter; defined in Sections 4.1 and 4.5). This is the declarative model extended with explicit threads and by-need computation. This model keeps most of the nice properties of the declarative model, e.g., reasoning is almost as simple, while being truly concurrent. This model can do both data-driven and demand-driven concurrency. It subsumes lazy functional programming and deterministic concurrent logic programming. Components interact by binding and using sharing dataflow variables.

- **Declarative model with exceptions** (defined in Sections 2.6.2 and 4.9.1). The concurrent declarative model with exceptions is no longer declarative, since programs can be written that expose nondeterminism.

- **Message-passing concurrent model** (see Chapter 5). This is the declarative model extended with communication channels (ports). This removes the limitation of the declarative concurrent model that it cannot implement programs with some nondeterminism, e.g., a client/server where several clients talk to a server. This is a useful generalization of the declarative concurrent model that is easy to program in and allows to restrict the nondeterminism to small parts of the program.

- **Stateful model** (see Chapters 6–7; defined in Section 6.3). This is the declarative model extended with explicit state. This model can express sequential object-oriented programming. A state is a sequence of values that is extended as the computation proceeds. Having explicit state means that a component does not always give the same result when called with the same arguments. The component can "remember" information from one call to the next. This allows the component to have a "history", which lets it interact more meaningfully with its environment by adapting and learning from its past. Reasoning with this model requires reasoning on the history.

- **Shared-state concurrent model** (see Chapter 8; defined in Section 8.1). This is the declarative model extended with both explicit state and threads. This model contains concurrent object-oriented programming. The concurrency is more expressive than the declarative concurrent model since it can use explicit state to wait simultaneously on one of several events occurring (this is called *nondeterministic choice*). Reasoning with this model is the

most complex since there can be multiple histories interacting in unpredictable ways.

- **Relational model** (in Chapter 9; defined in Section 9.1). This is the declarative model extended with search (which is sometimes called "don't know" nondeterminism, although the search algorithm is almost always deterministic). In the program, the search is expressed as sequence of choices. The search space is explored by making different choices until the result is satisfactory. This model allows to program with relations. It encompasses nondeterministic logic programming in the Prolog style. This model is a precursor to constraint programming, which is introduced in Chapter 12.

Later on, we will devote whole chapters to each of these models to explain what they are good for, how to program in them, and how to reason with them.

4.7.7 Using different models together

Typically, any well-written program of reasonable size has different parts written in different models. There are many ways to use different models together. This section gives an example of a particularly useful technique, which we call *impedance matching*, that naturally leads to using different models together in the same program.

Impedance matching is one of the most powerful and practical ways to implement the general principle of separation of concerns. Consider two computation models `Big` and `Small`, such that model `Big` is more expressive than `Small`, but harder to reason in. For example, model `Big` could be stateful and model `Small` declarative. With impedance matching, we can write a program in model `Small` that can live in the computational environment of model `Big`.

Impedance matching works by building an abstraction in model `Big` that is parameterized with a program in model `Small`. The heart of impedance matching is finding and implementing the right abstraction. This hard work only needs to be done once; afterwards there is only the easy work of using the abstraction. Perhaps surprisingly, it turns out that it is almost always possible to find and implement an appropriate abstraction. Here are some typical cases of impedance matching:

- Using a sequential component in a concurrent model. For example, the abstraction can be a *serializer* that accepts concurrent requests, passes them sequentially, and returns the replies correctly. Figure 4.35 gives an illustration. Plugging a sequential component into the serializer gives a concurrent component.

- Using a declarative component in a stateful model. For example, the abstraction can be a *storage manager* that passes its content to the declarative program and stores the result as its new content.

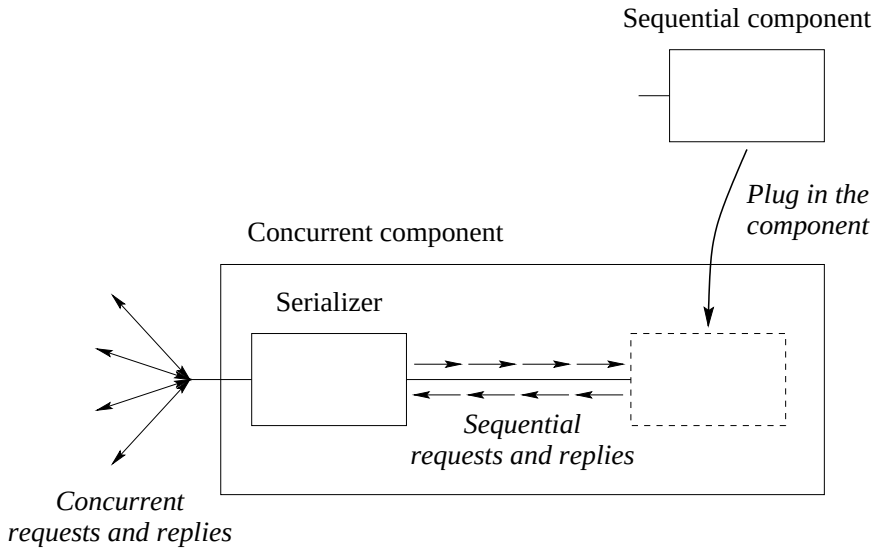

Figure 4.35: Impedance matching: example of a serializer

- Using a centralized component in a distributed model. A distributed model executes over more than one operating system process. For example, the abstraction can be a *collector* that accepts requests from any site and passes them to a single site.

- Using a component that is intended for a secure model in an insecure model. A insecure model is one that assumes the existence of malicious entities that can disturb programs in well-defined ways. A secure model assumes that no such entities exist. The abstraction can be a *protector* that insulates a computation by verifying all requests from other computations. The abstraction handles all the details of coping with the presence of malicious adversaries. A firewall is a kind of protector.

- Using a component that is intended for a closed model in an open model. An open model is one that lets independent computations find each other and interact. A closed model assumes that all computations are initially known. The abstraction can be a *connector* that accepts requests from one computation to connect to another, using an open addressing scheme.

- Using a component that assumes a reliable model in a model with partial failure. Partial failure occurs in a distributed system when part of the system fails. For example, the abstraction can be a *replicator* that implements fault tolerance by doing active replication between several sites and managing recovery when one fails.

These cases are orthogonal. As the examples show, it is often a good idea to implement several cases in one abstraction. This book has abstractions that illustrate all these cases and more. Usually, the abstraction puts a minor condition

on the program written in model `Small`. For example, a replicator often assumes that the function it is replicating is deterministic.

Impedance matching is extensively used in the Erlang project at Ericsson [9]. A typical Erlang abstraction takes a declarative program written in a functional language and makes it stateful, concurrent, and fault tolerant.

4.8 The Haskell language

We give a brief introduction to Haskell, a popular functional programming language supported by a number of interpreters and compilers [85, 148].[19] It is perhaps the most successful attempt to define a practical, completely declarative language. Haskell is a non-strict, strongly-typed functional language that supports currying and the monadic programming style. Strongly typed means that the types of all expressions are computed at compile time and all function applications must be type correct. The monadic style is a set of higher-order programming techniques that can be used to replace explicit state in many cases. The monadic style can do much more than just simulate state; we do not explain it in this brief introduction but we refer to any of the many papers and tutorials written about it [86, 209, 135].

Before giving the computation model, let us start with a simple example. We can write a factorial function in Haskell as follows:

```
factorial :: Integer -> Integer
factorial 0         = 1
factorial n | n > 0 = n * factorial (n-1)
```

The first line is the *type signature*. It specifies that `factorial` is a function that expects an argument of type `Integer` and returns a result of type `Integer`. Haskell does *type inferencing*, i.e., the compiler is able to automatically infer the type signatures, for almost all functions.[20] This happens even when the type signature is provided: the compiler then checks that the signature is accurate. Type signatures provide useful documentation.

The next two lines are the code for `factorial`. In Haskell a function definition can consist of many equations. To apply a function to an argument we do pattern matching; we examine the equations one by one from top to bottom until we find the first one whose pattern matches the argument. The first line of `factorial` only matches an argument of 0; in this case the answer is immediate, namely 1. If the argument is nonzero we try to match the second equation. This equation has a Boolean guard which must be true for the match to succeed. The second equation matches all arguments that are greater than 0; in that case we evaluate `n * factorial (n-1)`. What happens if we apply `factorial` to a negative

[19] The author of this section is Kevin Glynn.

[20] Except in a very few special cases which are beyond the scope of this section, such as polymorphic recursion.

argument? None of the equations match and the program will give a run-time error.

4.8.1 Computation model

A Haskell program consists of a single expression. This expression may contain many reducible subexpressions. In which order should they be evaluated? Haskell is a *non-strict* language, so no expression should be evaluated unless its result is definitely needed. Intuitively then, we should first reduce the leftmost expression until it is a function, substitute arguments in the function body (without evaluating them!) and then reduce the resulting expression. This evaluation order is called *normal order*. For example, consider the following expression:

```
(if n >= 0 then factorial else error) (factorial (factorial n))
```

This uses `n` to choose which function, `factorial` or `error`, to apply to the argument `(factorial (factorial n))`. It is pointless evaluating the argument until we have evaluated the `if then else` statement. Once this is evaluated we can substitute `factorial (factorial n)` in the body of `factorial` or `error` as appropriate and continue evaluation.

Let us explain in a more precise way how expressions reduce in Haskell. Imagine the expression as a tree.[21] Haskell first evaluates the leftmost subexpression until it evaluates to a data constructor or function:

- If it evaluates to a data constructor then evaluation is finished. Any remaining subexpressions remain unevaluated.
- If it evaluates to a function and it is not applied to any arguments then evaluation is finished.
- Otherwise, it evaluates to a function and is applied to arguments. Apply the function to the first argument (without evaluating it) by substituting it in the body of the function and re-evaluate.

Built-in functions such as addition and pattern matching cause their arguments to be evaluated before they can evaluate. For declarative programs this evaluation order has the nice property that it always terminates if any evaluation order could.

4.8.2 Lazy evaluation

Since arguments to functions are not automatically evaluated before function calls, we say that function calls in Haskell are *non-strict*. Although not mandated by the Haskell language, most Haskell implementations are in fact *lazy*, that is,

[21] For efficiency reasons, most Haskell implementations represent expressions as graphs, i.e., shared expressions are only evaluated once.

they ensure that expressions are evaluated at most once. The differences between lazy and non-strict evaluation are explained in Section 4.9.2.

Optimising Haskell compilers perform an analysis called *strictness analysis* to determine when the laziness is not necessary for termination or resource control. Functions that do not need laziness are compiled as eager ("strict") functions, which is much more efficient.

As an example of laziness we reconsider the calculation of a square root by Newton's method given in Section 3.2. The idea is that we first create an "infinite" list containing better and better approximations to the square root. We then traverse the list until we find the first approximation which is accurate enough and return it. Because of laziness we will only create as much of the list of approximations as we need.

```
sqrt x = head (dropWhile (not . goodEnough) sqrtGuesses)
   where
      goodEnough guess = (abs (x - guess*guess))/x < 0.00001
      improve guess = (guess + x/guess)/2.0
      sqrtGuesses = 1:(map improve sqrtGuesses)
```

The definitions following the `where` keyword are *local* definitions, i.e., they are only visible within `sqrt`. `goodEnough` returns true if the current guess is close enough. `improve` takes a guess and returns a better guess. `sqrtGuesses` produces the infinite list of approximations. The colon `:` is the list constructor, equivalent to `|` in Oz. The first approximation is 1. The following approximations are calculated by applying the `improve` function to the list of approximations. `map` is a function that applies a function to all elements of a list, similar to `Map` in Oz.[22] So the second element of `sqrtGuesses` will be `improve 1`, the third element will be `improve (improve 1)`. To calculate the n^{th} element of the list we evaluate `improve` on the $(n-1)^{th}$ element.

The expression `dropWhile (not . goodEnough) sqrtGuesses` drops the approximations from the front of the list that are not close enough. `(not . goodEnough)` is a *function composition*. It applies `goodEnough` to the approximation and then applies the boolean function `not` to the result. So `(not . goodEnough)` is a function that returns true if `goodEnough` returns false.

Finally, `head` returns the first element of the resulting list, which is the first approximation that was close enough. Notice how we have separated the calculation of the approximations from the calculation that chooses the appropriate answer.

4.8.3 Currying

From the reduction rules we see that a function that expects multiple arguments is actually applied to its arguments one at a time. In fact, applying an n-argument

[22] Note that the function and list arguments appear in a different order in the Haskell and Oz versions.

function to a single argument evaluates to an $(n-1)$ argument function specialized to the value of the first argument. This process is called *currying* (see also Section 3.6.6). We can write a function which doubles all the elements in a list by calling `map` with just one argument:

```
doubleList = map (\x -> 2*x)
```

The notation `\x -> 2*x` is Haskell's notation for an anonymous function (a λ expression). In Oz the same expression would be written **`fun {$ X} 2*X end`**. Let us see how `doubleList` evaluates:

```
doubleList [1,2,3,4]
 =>  map (\x -> 2*x) [1,2,3,4]
 =>  [2,4,6,8]
```

Note that list elements are separated by commas in Haskell.

4.8.4 Polymorphic types

All Haskell expressions have a statically-determined type. However, we are not limited to Haskell's predefined types. A program can introduce new types. For example, we can introduce a new type `BinTree` for binary trees:

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
```

A `BinTree` is either `Empty` or a `Node` consisting of an element and two subtrees. `Empty` and `Node` are data constructors: they build data structures of type `BinTree`. In the definition `a` is a type variable and stands for an arbitrary type, the type of elements contained in the tree. `BinTree Integer` is then the type of binary trees of integers. Notice how in a `Node` the element and the elements in subtrees are restricted to have the same type. We can write a `size` function that returns the number of elements in a binary tree as follows:

```
size :: BinTree a -> Integer
size Empty            = 0
size (Node val lt rt) = 1 + (size lt) + (size rt)
```

The first line is the type signature. It can be read as "For all types `a`, `size` takes an argument of type `BinTree a` and returns an `Integer`". Since `size` works on trees containing any type of element it is called a *polymorphic* function. The code for the function consists of two lines. The first line matches trees that are empty, their size is 0. The second line matches trees that are non-empty, their size is 1 plus the size of the left subtree plus the size of the right subtree.

Let us write a `lookup` function for an ordered binary tree. The tree contains tuples consisting of an integer key and a string value. It has type `BinTree (Integer,String)`. The lookup function returns a value with type `Maybe String`. This value will be `Nothing` if the key does not exist in the tree and `Just val` if `(k,val)` is in the tree:

```
lookup :: Integer -> BinTree (Integer,String) -> Maybe String
lookup k Empty = Nothing
lookup k (Node (nk,nv) lt rt) | k == nk = Just nv
lookup k (Node (nk,nv) lt rt) | k < nk  = lookup k lt
lookup k (Node (nk,nv) lt rt) | k > nk  = lookup k rt
```

At first sight, the type signature of `lookup` may look strange. Why is there a `->` between the `Integer` and tree arguments? This is due to currying. When we apply `lookup` to an integer key we get back a *new function* which when applied to a binary tree always looks up the same key.

4.8.5 Type classes

A disadvantage of the above definition of `lookup` is that the given type is very restrictive. We would like to make it polymorphic as we did with `size`. Then the same code could be used to search trees containing tuples of almost any type. However, we must restrict the first element of the tuple to be a type that supports the comparison operations `==`, `<`, and `>` (e.g., there is not a computable ordering for functions, so we do not want to allow functions as keys).

To support this Haskell has *type classes*. A type class gives a name to a group of functions. If a type supports those functions we say the type is a member of that type class. In Haskell there is a built in type class called `Ord` which supports `==`, `<`, and `>`. The following type signature specifies that the type of the tree's keys must be in type class `Ord`:

```
lookup :: (Ord a) => a -> BinTree (a,b) -> Maybe b
```

and indeed this is the type Haskell will infer for `lookup`. Type classes allow function names to be *overloaded*. The `<` operator for `Integers` is not the same as the `<` operator for `Strings`. Since a Haskell compiler knows the types of all expressions, it can substitute the appropriate type specific operation at each use. Type classes are supported by functional languages such as Clean and Mercury. (Mercury is a logic language with functional programming support.) Other languages, including Standard ML and Oz, can achieve a similar overloading effect by using functors.

Programmers can add their own types to type classes. For example, we could add the `BinTree` type to `Ord` by providing appropriate definitions for the comparison operators. If we created a type for complex numbers we could make it a member of the numeric type class `Num` by providing appropriate numerical operators. The most general type signature for `factorial` is

```
factorial :: (Num a, Ord a) => a -> a
```

So `factorial` can be applied to an argument of any type supporting numerical and comparison operations, returning a value of the same type.

⟨s⟩ ::=	
skip	Empty statement
⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
local ⟨x⟩ **in** ⟨s⟩ **end**	Variable creation
⟨x⟩$_1$=⟨x⟩$_2$	Variable-variable binding
⟨x⟩=⟨v⟩	Value creation
if ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Conditional
case ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Pattern matching
{⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$}	Procedure application
thread ⟨s⟩ **end**	Thread creation
{ByNeed ⟨x⟩ ⟨y⟩}	Trigger creation
try ⟨s⟩$_1$ **catch** ⟨x⟩ **then** ⟨s⟩$_2$ **end**	Exception context
raise ⟨x⟩ **end**	Raise exception
{FailedValue ⟨x⟩ ⟨y⟩}	**Failed value**

Table 4.3: The declarative concurrent kernel language with exceptions

4.9 Advanced topics

4.9.1 The declarative concurrent model with exceptions

In Section 2.6 we added exceptions to sequential declarative programming. Let us now see what happens when we add exceptions to concurrent declarative programming. We first explain how exceptions interact with concurrency. Then we explain how exceptions interact with by-need computation.

Exceptions and concurrency

So far, we have ignored exceptions in concurrent declarative programming. There is a very simple reason for this: if a component raises an exception in the declarative concurrent model then the model is no longer declarative! Let us add exceptions to the declarative concurrent model and see what happens. For the data-driven model, the resulting kernel language is given in Table 4.3. This table contains the **thread** and ByNeed instructions, the **try** and **raise** statements, and also one new operation, FailedValue, which handles the interaction between exceptions and by-need computation. We first explain the interaction between concurrency and exceptions; we leave FailedValue to the next section.

Let us investigate how exceptions can make the model nondeclarative. There are two basic ways. First, to be declarative, a component has to be deterministic. If the statements X=1 and X=2 are executed concurrently, then execution is no longer deterministic: one of them will succeed and the other will raise an exception. In the store, X will be bound either to 1 or to 2; both cases are possible. This is a clear case of observable nondeterminism. The exception is a *witness* to

this; it is raised on unification failure, which means that there is potentially an observable nondeterminism. The exception is not a guarantee of this; for example executing `X=1` and `X=2` in order in the same thread will raise an exception, yet `X` is always bound to 1. But if there are *no* exceptions, then execution is surely deterministic and hence declarative.

A second way that an exception can be raised is when an operation cannot complete normally. This can be due to internal reasons, e.g., the arguments are outside the operation's domain (such as dividing by zero), or external reasons, e.g., the external environment has a problem (such as trying to open a file that does not exist). In both cases, the exception indicates that an operation was attempted outside of its specification. When this happens, all bets are off, so to speak. From the viewpoint of semantics, there is no guarantee on what the operation has done; it could have done anything. Again, the operation has potentially become nondeterministic.

To summarize, when an exception is raised, this is an indication either of nondeterministic execution or of an execution outside specification. In either case, the component is no longer declarative. We say that the declarative concurrent model is declarative *modulo exceptions*. It turns out that the declarative concurrent model with exceptions is similar to the shared-state concurrent model of Chapter 8. This is explained in Section 8.1.

So what do we do when an exception occurs? Are we completely powerless to write a declarative program? Not at all. In some cases, the component can "fix things" so that it is still declarative when viewed from the outside. The basic problem is to make the component deterministic. All sources of nondeterminism have to be hidden from the outside. For example, if a component executes `X=1` and `X=2` concurrently, then the minimum it has to do is (1) catch the exception by putting a **`try`** around each binding, and (2) encapsulate `X` so its value is not observable from the outside. See the failure confinement example in Section 4.1.4.

Exceptions and by-need computation

In Section 2.6, we added exceptions to the declarative model as a way to handle abnormal conditions without encumbering the code with error checks. If a binding fails, it raises a failure exception, which can be caught and handled by another part of the application.

Let us see how to extend this idea to by-need computation. What happens if the execution of a by-need trigger cannot complete normally? In that case it does not calculate a value. For example:

```
X={ByNeed fun {$} A=foo(1) B=foo(2) in A=B A end}
```

What should happen if a thread needs `X`? Triggering the calculation causes a failure when attempting the binding `A=B`. It is clear that `X` cannot be bound to a value, since the by-need computation is not able to complete. On the other hand, we cannot simply leave `X` unbound since the thread that needs `X` expects a value. The right solution is for that thread to raise an exception. To ensure this,

we can bind `X` to a special value called a *failed value*. Any thread that needs a failed value will raise an exception.

We extend the kernel language with the operation `FailedValue`, which creates a failed value:

```
X={FailedValue cannotCalculate}
```

Its definition is given in the supplements file on the book's Web site. It creates a failed value that encapsulates the exception `cannotCalculate`. Any thread that attempts to use `X` will raise the exception `cannotCalculate`. Any partial value can be encapsulated inside the failed value.

With `FailedValue` we can define a "robust" version of `ByNeed` that automatically creates a failed value when a by-need computation raises an exception:

```
proc {ByNeed2 P X}
   {ByNeed
      proc {$ X}
         try Y in {P Y} X=Y
         catch E then X={FailedValue E} end
      end X}
end
```

`ByNeed2` is called in the same way as `ByNeed`. If there is any chance that the by-need computation will raise an exception, then `ByNeed2` will encapsulate the exception in a failed value.

Table 4.3 gives the kernel language for the complete declarative concurrent model including both by-need computation and exceptions. The kernel language contains the operations `ByNeed` and `FailedValue` as well as the **`try`** and **`raise`** statements. The operation `{FailedValue` ⟨x⟩ ⟨y⟩`}` encapsulates the exception ⟨x⟩ in the failed value ⟨y⟩. Whenever a thread needs ⟨y⟩, the statement **`raise`** ⟨x⟩ **`end`** is executed in the thread.

One important use of failed values is in the implementation of dynamic linking. Recall that by-need computation is used to load and link modules on need. If the module could not be found, then the module reference is bound to a failed value. Then, whenever a thread tries to use the nonexistent module, an exception is raised.

4.9.2 More on lazy execution

There is a rich literature on lazy execution. In Section 4.5 we have just touched the tip of the iceberg. Let us now continue the discussion of lazy execution. We bring up two topics:

- **Language design issues**. When designing a new language, what is the role of laziness? We briefly summarize the issues involved.
- **Reduction order and parallelism**. Modern functional programming languages, as exemplified by Haskell, often use a variant of laziness called

non-strict evaluation. We give a brief overview of this concept and why it is useful.

Language design issues

Should a declarative language be lazy or eager or both? This is part of a larger question: should a declarative language be a subset of an extended, nondeclarative language? Limiting a language to one computation model allows to optimize its syntax and semantics for that model. For programs that "fit" the model, this can give amazingly concise and readable code. Haskell and Prolog are particularly striking examples of this approach to language design [17, 182]. Haskell uses lazy evaluation throughout and Prolog uses Horn clause resolution throughout. See Section 4.8 and Section 9.7, respectively, for more information on these two languages. FP, an early and influential functional language, carried this to an extreme with a special character set, which paradoxically reduces readability [12]. However, as we shall see in Section 4.7, many programs require more than one computation model. This is true also for lazy versus eager execution. Let us see why:

- For programming in the small, e.g., designing algorithms, eagerness is important when execution complexity is an issue. Eagerness makes it easy to design and reason about algorithms with desired worst-case complexities. Laziness makes this much harder; even experts get confused. On the other hand, laziness is important when designing algorithms with persistence, i.e., that can have multiple coexisting versions. Section 4.5.8 explains why this is so and gives an example. We find that a good approach is to use eagerness by default and to put in laziness explicitly, exactly where it is needed. Okasaki does this with a version of the eager functional language Standard ML extended with explicit laziness [138].

- For programming in the large, eagerness and laziness both have important roles when interfacing components. For example, consider a pipeline communication between a producer and consumer component. There are two basic ways to control this execution: either the producer decides when to calculate new elements ("push" style) or the consumer asks for elements as it needs them ("pull" style). A push style implies an eager execution and a pull style implies a lazy execution. Both styles can be useful. For example, a bounded buffer enforces a push style when it is not full and a pull style when it is full.

We conclude that a declarative language intended for general-purpose programming should support both eager and lazy execution, with eager being the default and lazy available through a declaration. If one is left out, it can always be encoded, but this makes programs unnecessarily complex.

Reduction order and parallelism

We saw that lazy evaluation will evaluate a function's arguments only when they are needed. Technically speaking, this is called *normal order reduction*. When executing a declarative program, normal order reduction will always choose to reduce first the leftmost expression. After doing one reduction step, then again the leftmost expression is chosen. Let us look at an example to see how this works. Consider the function `F1` defined as follows:

```
fun {F1 A B}
   if B then A else 0 end
end
```

Let us evaluate the expression `{F1 {F2 X} {F3 Y}}`. The first reduction step applies `F1` to its arguments. This substitutes the arguments into the body of `F1`. This gives `if {F3 Y} then {F2 X} else 0 end`. The second step starts the evaluation of `F3`. If this returns `false`, then `F2` is not evaluated at all. We can see intuitively that normal order reduction only evaluates expressions when they are needed.

There are many possible reduction orders. This is because every execution step gives a choice which function to reduce next. With declarative concurrency, many of these orders can appear during execution. This makes no difference in the result of the calculation: we say that there is no observable nondeterminism.

Besides normal order reduction, there is another interesting order called *applicative order reduction*. It always evaluates a function's arguments before evaluating the function. This is the same as eager evaluation. In the expression `{F1 {F2 X} {F3 Y}}`, this evaluates both `{F2 X}` and `{F3 Y}` before evaluating `F1`. With applicative order reduction, if either `{F2 X}` or `{F3 Y}` goes into an infinite loop, then the whole computation will go into an infinite loop. This is true even though the results of `{F2 X}` or `{F3 Y}` might not be needed by the rest of the computation. We say that applicative order reduction is *strict*.

For all declarative programs, we can prove that all reduction orders that terminate give the same result. This result is a consequence of the Church-Rosser Theorem, which shows that reduction in the λ calculus is *confluent*, i.e., reductions that start from the same expression and follow different paths can always be brought back together again. We can say this another way: changing the reduction order only affects whether or not the program terminates but does not change its result. We can also prove that normal order reduction gives the smallest number of reduction steps when compared to any other reduction order.

Non-strict evaluation A functional programming language whose computation model terminates when normal order reduction terminates is called a *non-strict language*. We mention non-strict evaluation because it is used in Haskell, a popular functional language. The difference between non-strict and lazy evaluation is subtle. A lazy language does the absolute minimum number of reduction steps. A non-strict language might do more steps, but it is still guaranteed to

	Asynchronous	Synchronous
Send	bind a variable	wait until variable needed
Receive	use variable immediately	wait until variable bound

Table 4.4: Dataflow variable as communication channel

terminate in those cases when the lazy language terminates. To better see the difference between lazy and non-strict, consider the following example:

```
local X={F 4} in X+X end
```

In a non-strict language `{F 4}` may be computed twice. In a lazy language `{F 4}` will be computed exactly once when `X` is first needed and the result reused for each subsequent occurrence of `X`. A lazy language is always non-strict, but not the other way around.

The difference between non-strict and lazy evaluation becomes important in a parallel processor. For example, during the execution of `{F1 {F2 X} {F3 Y}}` we might start executing `{F2 X}` on an available processor, even before we know whether it is really needed or not. This is called *speculative execution*. If later on we find out that `{F2 X}` is needed, then we have a head start in its execution. If `{F2 X}` is not needed, then we abort it as soon as we know this. This might waste some work, but since it is on another processor it will not cause a slowdown. A non-strict language can be implemented with speculative execution.

Non-strictness is problematic when we want to extend a language with explicit state (as we will do in Chapter 6). A non-strict language is hard to extend with explicit state because non-strictness introduces a fundamental unpredictability in a language's execution. We can never be sure how many times a function is evaluated. In a declarative model this is not serious since it does not change computations' results. It becomes serious when we add explicit state. Functions with explicit state can have unpredictable results. Lazy evaluation has the same problem but to a lesser degree: evaluation order is data-dependent but at least we know that a function is evaluated at most once. The solution used in the declarative concurrent model is to make eager evaluation the default and lazy evaluation require an explicit declaration. The solution used in Haskell is more complicated: to avoid explicit state and instead use a kind of accumulator called a *monad*. The monadic approach uses higher-order programming to make the state threading implicit. The extra arguments are part of function inputs and outputs. They are threaded by defining a new function composition operator.

4.9.3 Dataflow variables as communication channels

In the declarative concurrent model, threads communicate through shared dataflow variables. There is a close correspondence between operations on dataflow variables and operations on a communication channel. We consider a dataflow variable as a kind of communication channel and a thread as a kind of object. Then

binding a variable is a kind of send and waiting until a variable is bound is a kind of receive. The channel has the property that only one message can be sent but the message can be received many times. Let us investigate this analogy further.

On a communication channel, send and receive operations can be asynchronous or synchronous. This gives four possibilities in all. Can we express these possibilities with dataflow variables? Two of the possibilities are straightforward since they correspond to a standard use of dataflow execution:

- Binding a variable corresponds to an asynchronous send. The binding can be done independent of whether any threads have received the message.
- Waiting until a variable is bound corresponds to a synchronous receive. The binding must exist for the thread to continue execution.

What about asynchronous receive and synchronous send? In fact, they are both possible:

- Asynchronous receive means simply to use a variable before it is bound. For example, the variable can be inserted in a data structure before it is bound. Of course, any operation that needs the variable's value will wait until the value arrives.
- Synchronous send means to wait with binding until the variable's value is received. Let us consider that a value is received if it is needed by some operation. Then the synchronous send can be implemented with by-need triggers:

```
proc {SyncSend X M}
Sync in
   {ByNeed proc {$ _} X=M Sync=unit end X}
   {Wait Sync}
end
```

  Doing `{SyncSend X M}` sends `M` on channel `X` and waits until it has been received.

Table 4.4 summarizes these four possibilities.

Communication channels sometimes have nonblocking send and receive operations. These are not the same as asynchronous operations. The defining characteristic of a *nonblocking operation* is that it returns immediately with a boolean result telling whether the operation was successful or not. With dataflow variables, a nonblocking send is trivial since a send is always successful. A nonblocking receive is more interesting. It consists in checking whether the variable is bound or not, and returning **`true`** or **`false`** accordingly. This can be implemented with the `IsDet` function. `{IsDet X}` returns immediately with **`true`** if `X` is bound and with **`false`** otherwise. To be precise, `IsDet` returns true if `X` is *determined*, i.e., bound to a number, record, or procedure. Needless to say, `IsDet` is not a declarative operation.

4.9.4 More on synchronization

We have seen that threads can communicate through shared dataflow variables. When a thread needs the result of a calculation done by another thread then it waits until this result is available. We say that it *synchronizes* on the availability of the result. Synchronization is one of the fundamental concepts in concurrent programming. Let us now investigate this concept more closely.

We first define precisely the basic concept, called a *synchronization point*. Consider threads T1 and T2, each doing a sequence of computation steps. T1 does $\alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow$ ... and T2 does $\beta_0 \rightarrow \beta_1 \rightarrow \beta_2 \rightarrow$ .... The threads actually execute together in one global computation. This means that there is one global sequence of computation steps that contains the steps of each thread, interleaved: $\alpha_0 \rightarrow \beta_0 \rightarrow \beta_1 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow$ .... There are many ways that the two computations can be interleaved. But not all interleavings can occur in real computations:

- Because of fairness, it is not possible to have an infinite sequence of α steps without some β steps. Fairness is a global property that is enforced by the system.
- If the threads depend on each other's results in some way, then there are additional constraints called *synchronization points*. A synchronization point links two computation steps β_i and α_j. We say that β_i *synchronizes on* α_j if in every interleaving that can occur in a real computation, β_i occurs *after* α_j. Synchronization is a local property that is enforced by operations happening in the threads.

How does the program specify when to synchronize? There are two broad approaches:

- **Implicit synchronization**. In this approach, the synchronization operations are not visible in the program text; they are part of the operational semantics of the language. For example, using a dataflow variable will synchronize on the variable being bound to a value.
- **Explicit synchronization**. In this approach, the synchronization operations are visible in the program text; they consist of explicit operations put there by the programmer. For example, Section 4.3.3 shows a demand-driven producer/consumer that uses a programmed trigger. Later on in the book we will see other ways to do explicit synchronization, for example by using locks or monitors (see Chapter 8).

There are two directions of synchronization:

- **Supply-driven synchronization (eager execution)**. Attempting to execute an operation causes the operation to wait until its arguments are available. In other words, the operation synchronizes on the availability of

	Supply-driven	Demand-driven
Implicit	dataflow execution	lazy execution
Explicit	locks, monitors, etc.	programmed trigger

Table 4.5: Classifying synchronization

its arguments. This waiting has no effect on whether or not the arguments will be calculated; if some other thread does not calculate them then the operation will wait indefinitely.

- **Demand-driven synchronization (lazy execution)**. Attempting to execute an operation causes the calculation of its arguments. In other words, the calculation of the arguments synchronizes on the operation needing them.

Table 4.5 shows the four possibilities that result. All four are practical and exist in real systems. Explicit synchronization is the primary mechanism in most languages that are based on a stateful model, e.g., Java, Smalltalk, and C++. This mechanism is explained in Chapter 8. Implicit synchronization is the primary mechanism in most languages that are based on a declarative model, e.g., functional languages such as Haskell use lazy evaluation and logic languages such as Prolog and concurrent logic languages use dataflow execution. This mechanism is presented in this chapter.

All four possibilities can be used efficiently in the computation models of this book. This lets us compare their expressiveness and ease of use. We find that concurrent programming is simpler with implicit synchronization than with explicit synchronization. In particular, we find that programming with dataflow execution makes concurrent programs simpler. Even in a stateful model, like the one in Chapter 8, dataflow execution is advantageous. After comparing languages with explicit and implicit synchronization, Bal *et al* come to the same conclusion: that dataflow variables are "spectacularly expressive" in concurrent programming as compared to explicit synchronization, even without explicit state [14]. This expressiveness is one of the reasons why we emphasize implicit synchronization in the book. Let us now examine more closely the usefulness of dataflow execution.

4.9.5 Usefulness of dataflow variables

Section 4.2.3 shows how dataflow execution is used for synchronization in the declarative concurrent model. There are many other uses for dataflow execution. This section summarizes these uses. We give pointers to examples throughout the book to illustrate them. Dataflow execution is useful because:

- It is a powerful primitive for concurrent programming (see this chapter and Chapter 8). It can be used for synchronizing and communicating between

concurrent computations. Many concurrent programming techniques become simplified and new techniques become possible when using dataflow variables.

- It removes order dependencies between parts of a program (see this chapter and Chapter 8). To be precise, it replaces static dependencies (decided by the programmer) by dynamic dependencies (decided by the data). This is the basic reason why dataflow computation is useful for parallel programming. The output of one part can be passed directly as input to the next part, independent of the order in which the two parts are executed. When the parts execute, the second one will block only if necessary, i.e., only if it needs the result of the first and it is not yet available.

- It is a powerful primitive for distributed programming (see Chapter 11). It improves latency tolerance and third-party independence. A dataflow variable can be passed among sites arbitrarily. At all times, it "remembers its origins," i.e., when the value becomes known then the variable will receive it. The communication needed to bind the variable is part of the variable and not part of the program manipulating the variable.

- It makes it possible to do declarative calculations with partial information. This was exploited in Chapter 3 with difference lists. One way to look at partial values is as complete values that are only partially known. This is a powerful idea that is further exploited in constraint programming (see Chapter 12).

- It allows the declarative model to support logic programming (see Section 9.3). That is, it is possible to give a logical semantics to many declarative programs. This allows reasoning about these programs at a very high level of abstraction. From a historical viewpoint, dataflow variables were originally discovered in the context of concurrent logic programming, where they are called *logic variables*.

An insightful way to understand dataflow variables is to see them as a middle ground between having no state and having state:

- A dataflow variable is stateful, because it can change state (i.e., be bound to a value), but it can be bound to just one value in its lifetime. The stateful aspect can be used to get some of the advantages of programming with state (as explained in Chapter 6) while staying within a declarative model. For example, difference lists can be appended in constant time, which is not possible for lists in a pure functional model.

- A dataflow variable is stateless, because binding is monotonic. By *monotonic* we mean that more information can be added to the binding, but no information can be changed or removed. Assume the variable is bound to a

partial value. Later on, more and more of the partial value can be bound, which amounts to binding the unbound variables inside the partial value. But these bindings cannot be changed or undone.

The stateless aspect can be used to get some of the advantages of declarative programming within a non-declarative model. For example, it is possible to add concurrency to the declarative model, giving the declarative concurrent model of this chapter, precisely because threads communicate through shared dataflow variables.

Futures and I-structures

The dataflow variables used in this book are but one technique to implement dataflow execution. Another, quite popular technique is based on a slightly different concept, the *single-assignment variable*. This is a mutable variable that can be assigned only once. This differs from a dataflow variable in that the latter can be assigned (perhaps multiple times) to many partial values, as long as the partial values are compatible with each other.

Two of the best-known instances of the single-assignment variable are *futures* and *I-structures*. The purpose of futures and I-structures is to increase the potential parallelism of a program by removing inessential dependencies between calculations. They allow concurrency between a computation that calculates a value and one that uses the value. This concurrency can be exploited on a parallel machine. We define futures and I-structures and compare them with dataflow variables.

Futures were first introduced in Multilisp, a language intended for writing parallel programs [68]. Multilisp introduces the function call `(future E)` (in Lisp syntax), where E is any expression. This does two things: it immediately returns a placeholder for the result of E and it initiates a concurrent evaluation of E. When the value of E is needed, i.e., a computation tries to access the placeholder, then the computation blocks until the value is available. We model this as follows in the declarative concurrent model (where E is a zero-argument function):

```
fun {Future E}
X in
   thread X={E} end
   !!X
end
```

A future can only be bound by the concurrent computation that is created along with it. This is enforced by returning a read-only variable. Multilisp also has a `delay` construct that does not initiate any evaluation but uses by-need execution. It causes evaluation of its argument only when the result is needed.

An I-structure (for *incomplete structure*) is an array of single-assignment variables. Individual elements can be accessed before all the elements are computed. I-structures were introduced as a language construct for writing parallel programs

on dataflow machines, for example in the dataflow language Id [11, 202, 88, 131]. I-structures are also used in pH ("parallel Haskell"), a recent language design that extends Haskell for implicit parallelism [132, 133]. An I-structure permits concurrency between a computation that calculates the array elements and a computation that uses their values. When the value of an element is needed, then the computation blocks until it is available. Like a future and a read-only variable, an element of an I-structure can only be bound by the computation that calculates it.

There is a fundamental difference between dataflow variables on one side and futures and I-structures on the other side. The latter can be bound only once, whereas dataflow variables can be bound more than once, as long as the bindings are consistent with each other. Two partial values are *consistent* if they are unifiable. A dataflow variable can be bound many times to different partial values, as long as the partial values are unifiable. Section 4.3.1 gives an example when doing stream communication with multiple readers. Multiple readers are each allowed to bind the list's tail, since they bind it in a consistent way.

4.10 Historical notes

Declarative concurrency has a long and respectable history. We give some of the highlights. In 1974, Gilles Kahn defined a simple Algol-like language with threads that communicate by channels that behave like FIFO queues with blocking wait and nonblocking send [97]. He called this model *determinate parallel programming*.[23] In Kahn's model, a thread can wait on only one channel at a time, i.e., each thread always knows from what channel the next input will come. Furthermore, only one thread can send on each channel. This last restriction is actually a bit too strong. Kahn's model could be extended to be like the declarative concurrent model. More than one thread could send on a channel, as long as the sends are ordered deterministically. For example, two threads could take turns sending on the same channel.

In 1977, Kahn and David MacQueen extended Kahn's original model in significant ways [98]. The extended model is demand-driven, supports dynamic reconfiguration of the communication structure, and allows multiple readers on the same channel.

In 1990, Vijay Saraswat *et al* generalized Kahn's original model to concurrent constraints [164]. This adds partial values to the model and reifies communication channels as streams. Saraswat *et al* define first a *determinate concurrent constraint language*, which is essentially the same as the data-driven model of this chapter. It generalizes Kahn's original model to make possible programming techniques such as dynamic reconfiguration, channels with multiple readers, incomplete messages, difference structures, and tail-recursive append.

[23] By "parallelism" he means concurrency. In those days the term parallelism was used to cover both concepts.

Saraswat *et al* define the concept of *resting point*, which is closely related to partial termination as defined in Section 13.2. A resting point of a program is a store σ that satisfies the following property. When the program executes with this store, no information is ever added (the store is unchanged). The store existing when a program is partially terminated is a resting point.

The declarative concurrent models of this book have strong relationships to the papers cited above. The basic concept of determinate concurrency was defined by Kahn. The existence of the data-driven model is implicit in the work of Saraswat *et al*. The demand-driven model is related to the model of Kahn and MacQueen. The contribution of this book is to place these models in a uniform framework that subsumes all of them. Section 4.5 defines a demand-driven model by adding by-need synchronization to the data-driven model. By-need synchronization is based on the concept of *needing* a variable. Because need is defined as a monotonic property, this gives a quite general declarative model that has both concurrency and laziness.

4.11 Exercises

1. **Thread semantics**. Consider the following variation of the statement used in Section 4.1.3 to illustrate thread semantics:

```
local B in
   thread B=true end
   thread B=false end
   if B then {Browse yes} end
end
```

   For this exercise, do the following:

   (a) Enumerate all possible executions of this statement.

   (b) Some of these executions cause the program to terminate abnormally. Make a small change to the program to avoid these abnormal terminations.

2. **Threads and garbage collection**. This exercise examines how garbage collection behaves with threads and dataflow variables. Consider the following program:

```
proc {B _}
   {Wait _}
end

proc {A}
   Collectible={NewDictionary}
in
   {B Collectible}
```

```
end
```

After the call `{A}` is done, will `Collectible` become garbage? That is, will the memory occupied by `Collectible` be recovered? Give an answer by thinking about the semantics. Verify that the Mozart system behaves in this way.

3. **Concurrent Fibonacci**. Consider the following sequential definition of the Fibonacci function:

```
fun {Fib X}
   if X=<2 then 1
   else
      {Fib X-1}+{Fib X-2}
   end
end
```

and compare it with the concurrent definition given in Section 4.2.3. Run both on the Mozart system and compare their performance. How much faster is the sequential definition? How many threads are created by the concurrent call `{Fib N}` as a function of `N`?

4. **Order-determining concurrency**. Explain what happens when executing the following:

```
declare A B C D in
thread D=C+1 end
thread C=B+1 end
thread A=1 end
thread B=A+1 end
{Browse D}
```

In what order are the threads created? In what order are the additions done? What is the final result? Compare with the following:

```
declare A B C D in
A=1
B=A+1
C=B+1
D=C+1
{Browse D}
```

Here there is only one thread. In what order are the additions done? What is the final result? What do you conclude?

5. **The `Wait` operation**. Explain why the `{Wait X}` operation could be defined as:

```
proc {Wait X}
   if X==unit then skip else skip end
end
```

Use your understanding of the dataflow behavior of the **if** statement and == operation.

6. **Thread scheduling**. Section 4.7.3 shows how to skip over already-calculated elements of a stream. If we use this technique to sum the elements of the integer stream in Section 4.3.1, the result is *much* smaller than 11249925000, which is the sum of the integers in the stream. Why is it so much smaller? Explain this result in terms of thread scheduling.

7. **Programmed triggers using higher-order programming**. Programmed triggers can be implemented by using higher-order programming instead of concurrency and dataflow variables. The producer passes a zero-argument function `F` to the consumer. Whenever the consumer needs an element, it calls the function. This returns a pair `X#F2` where `X` is the next stream element and `F2` is a function that has the same behavior as `F`. Modify the example of Section 4.3.3 to use this technique.

8. **Dataflow behavior in a concurrent setting**. Consider the function `{Filter In F}`, which returns the elements of `In` for which the boolean function `F` returns **true**. Here is a possible definition of `Filter`:

```
fun {Filter In F}
   case In
   of X|In2 then
      if {F X} then X|{Filter In2 F}
      else {Filter In2 F} end
   else
      nil
   end
end
```

Executing the following:

```
{Show {Filter [5 1 2 4 0] fun {$ X} X>2 end}}
```

displays:

```
[5 4]
```

So `Filter` works as expected in the case of a sequential execution when all the input values are available. Let us now explore the dataflow behavior of `Filter`.

(a) What happens when we execute the following:

```
declare A
{Show {Filter [5 1 A 4 0] fun {$ X} X>2 end}}
```

One of the list elements is a variable `A` that is not yet bound to a value. Remember that the **`case`** and **`if`** statements will suspend the thread in which they execute, until they can decide which alternative path to take.

(b) What happens when we execute the following:

```
declare Out A
thread Out={Filter [5 1 A 4 0] fun {$ X} X>2 end} end
{Show Out}
```

Remember that calling `Show` displays its argument as it exists at the instant of the call. Several possible results can be displayed; which and why? Is the `Filter` function deterministic? Why or why not?

(c) What happens when we execute the following:

```
declare Out A
thread Out={Filter [5 1 A 4 0] fun {$ X} X>2 end} end
{Delay 1000}
{Show Out}
```

Remember that the call `{Delay N}` suspends its thread for at least `N` milliseconds. During this time, other ready threads can be executed.

(d) What happens when we execute the following:

```
declare Out A
thread Out={Filter [5 1 A 4 0] fun {$ X} X>2 end} end
thread A=6 end
{Delay 1000}
{Show Out}
```

What is displayed and why?

9. **Digital logic simulation**. In this exercise we will design a circuit to add n-bit numbers and simulate it using the technique of Section 4.3.5. Given two n-bit binary numbers, $(x_{n-1}...x_0)_2$ and $(y_{n-1}...y_0)_2$. We will build a circuit to add these numbers by using a chain of full adders, similar to doing long addition by hand. The idea is to add each pair of bits separately, passing the carry to the next pair. We start with the low-order bits x_0 and y_0. Feed them to a full adder with the third input $z = 0$. This gives a sum bit s_0 and a carry c_0. Now feed x_1, y_1, and c_0 to a second full adder. This gives a new sum s_1 and carry c_1. Continue this for all n bits. The final sum is $(s_{n-1}...s_0)_2$. For this exercise, program the addition circuit using full adders. Verify that it works correctly by feeding it several additions.

10. **Basics of laziness**. Consider the following program fragment: