

Why does this algorithm work? Consider any two nodes a and b with a path between them: $a \rightarrow n_1 \rightarrow n_2 \rightarrow \dots \rightarrow n_k \rightarrow b$ (where $k \geq 0$). We have to show that the final graph has an edge from a to b . The nodes n_1 through n_k are encountered in some order by the algorithm. When the algorithm encounters a node n_i , it “short circuits” the node, i.e., it creates a new path from a to b that avoids the node. Therefore when the algorithm has encountered all nodes, it has created a path that avoids all of them, i.e., it has an edge directly from a to b .

Representing a graph

To write up the algorithm as a program, we first have to choose a representation for directed graphs. Let us consider two possible representations:

- The **adjacency list** representation. The graph is a list with elements of the form $I\#Ns$ where I identifies a node and Ns is an ordered list of its immediate successors. As we will see below, ordered lists of successors are more efficient to calculate with than unordered lists.
- The **matrix** representation. The graph is a two-dimensional array. The element with coordinates (I,J) is true if there is an edge from node I to node J . Otherwise, the element is false.

We find that the choice of representation strongly influences what is the best computation model. In what follows, we assume that all graphs have at least one node and that the nodes are consecutive integers. We first give a declarative algorithm that uses the adjacency list representation [139]. We then give an in-place stateful algorithm that uses the matrix representation [41]. We then give a second declarative algorithm that also uses the matrix representation. Finally, we compare the three algorithms.

Converting between representations

To make comparing the two algorithms easier, we first define routines to convert from the adjacency list representation to the matrix representation and vice versa. Here is the conversion from adjacency list to matrix:

```

fun {L2M GL}
  M={Map GL fun {$ I#_} I end}
  L={FoldL M Min M.1}
  H={FoldL M Max M.1}
  GM={NewArray L H unit}
in
  for I#Ns in GL do
    GM.I:={NewArray L H false}
    for J in Ns do GM.I.J:=true end
  end
  GM
end

```

```

fun {DeclTrans G}
  Xs={Map G fun {$ X#_} X end}
in
  {FoldL Xs
   fun {$ InG X}
   SX={Succ X InG} in
     {Map InG
      fun {$ Y#SY}
      Y#if {Member X SY} then
        {Union SY SX} else SY end
      end}
   end G}
end

```

Figure 6.10: Transitive closure (first declarative version)

In this routine, as in all following routines, we use GL for the adjacency list representation and GM for the matrix representation. Here is the conversion from matrix to adjacency list:

```

fun {M2L GM}
  L={Array.low GM}
  H={Array.high GM}
in
  for I in L..H collect:C do
    {C I#for J in L..H collect:D do
      if GM.I.J then {D J} end
    end}
  end
end

```

This uses the loop syntax including the accumulation procedure `collect:C` to good advantage.

Declarative algorithm

We first give a declarative algorithm for transitive closure. The graph is represented as an adjacency list. The algorithm needs two utility routines, `Succ`, which returns the successor list of a given node, and `Union`, which calculates the union of two ordered lists. We develop the algorithm by successive transformation of the abstract algorithm. This design method is known as stepwise refinement.

The outermost loop in the abstract algorithm transforms the graph in successive steps. Our declarative algorithm does the same by using `FoldL`, which defines a loop with accumulator. This means we can define the main function `DeclTrans` as follows:

```

fun {DeclTrans G}
  Xs={Nodes G} in
    {FoldL Xs
      fun {$ InG X}
        SX={Succ X InG} in
          for each node Y in pred(X, InG):
            for each node Z in SX:
              add edge (Y, Z)
            end G}
    end

```

The next step is to implement the two inner loops:

```

for each node Y in pred(X, InG):
  for each node Z in SX:
    add edge (Y, Z)

```

These loops transform one graph into another, by adding edges. Since our graph is represented by a list, a natural choice is to use `Map`, which transforms one list into another. This gives the following code, where `Union` is used to add the successor list of `X` to that of `Y`:

```

{Map InG
  fun {$ Y#SY}
    Y#if "Y in pred(X, InG)" then
      {Union SY SX} else SY end
    end}

```

We finish up by noticing that `Y` is in `pred(X, InG)` if and only if `X` is in `succ(Y, InG)`. This means we can write the `if` condition as follows:

```

{Map InG
  fun {$ Y#SY}
    Y#if {Member X SY} then
      {Union SY SX} else SY end
    end}

```

Putting everything together we get the final definition in Figure 6.10. This uses `Map` to calculate `{Nodes G}`. We conclude by remarking that `FoldL`, `Map`, and other routines such as `Member`, `Filter`, etc., are basic building blocks that must be mastered when writing declarative algorithms.

To finish up our presentation of the declarative algorithm, we give the definitions of the two utility routines. `Succ` returns the list of successors of a node:

```

fun {Succ X G}
  case G of Y#SY|G2 then
    if X==Y then SY else {Succ X G2} end
  end
end

```

```

proc {StateTrans GM}
  L={Array.low GM}
  H={Array.high GM}
in
  for K in L..H do
    for I in L..H do
      if GM.I.K then
        for J in L..H do
          if GM.K.J then GM.I.J:=true end
        end
      end
    end
  end
end

```

Figure 6.11: Transitive closure (stateful version)

Succ assumes that x is always in the adjacency list, which is true in our case. Union returns the union of two sets, where all sets are represented as ordered lists:

```

fun {Union A B}
  case A#B
  of nil#B then B
  [] A#nil then A
  [] (X|A2)#(Y|B2) then
    if X==Y then X|{Union A2 B2}
    elseif X<Y then X|{Union A2 B}
    elseif X>Y then Y|{Union A B2}
    end
  end
end

```

Union's execution time is proportional to the length of the smallest input list because its input lists are ordered. If the lists were not ordered, its execution time would be proportional to the product of their lengths (why?), which is usually much larger.

Stateful algorithm

We give a stateful algorithm for transitive closure. The graph is represented as a matrix. This algorithm assumes that the matrix contains the initial graph. It then calculates the transitive closure in-place, i.e., by updating the input matrix itself. Figure 6.11 gives the algorithm. For each node K , this looks at each potential edge (I, J) and adds it if there is both an edge from I to K and from K to J . We show now the stepwise transformation that leads to this algorithm. We first restate the abstract algorithm with the proper variable names:

```

fun {DeclTrans2 GT}
  H={Width GT}
  fun {Loop K InG}
    if K=<H then
      G={MakeTuple g H} in
        for I in 1..H do
          G.I={MakeTuple g H}
          for J in 1..H do
            G.I.J=InG.I.J orelse (InG.I.K andthen InG.K.J)
          end
        end
      {Loop K+1 G}
    else InG end
  end
in
  {Loop 1 GT}
end

```

Figure 6.12: Transitive closure (second declarative version)

For each node k in the graph G :
 for each node i in $\text{pred}(k, G)$:
 for each node j in $\text{succ}(k, G)$:
 add the edge (i, j) to G .

This leads to the following refinement:

```

proc {StateTrans GM}
  L={Array.low GM}
  H={Array.high GM}
in
  for K in L..H do
    for I in L..H do
      if GM.I.K then
        for each J in  $\text{succ}(K, GM)$  do GM.I.J:=true
      end
    end
  end
end

```

We note that J is in $\text{succ}(K, GM)$ if $GM.K.J$ is true. This means we can replace the inner loop by:

```

for J in L..H do
  if GM.K.J then GM.I.J:=true end
end

```

Second declarative algorithm

Inspired by the stateful algorithm, we develop a second declarative algorithm. The second algorithm uses a series of tuples to store the successive approximations of the transitive closure. We use the variable `GT` instead of `GM` to emphasize this change in representation. A tuple is a record with fields numbered consecutively from 1 to a maximum value. So this algorithm is restricted to nodes whose numbering starts from 1. Note that `MakeTuple` creates a tuple with unbound variables. Figure 6.12 gives the algorithm.

This is somewhat more complicated than the stateful version. Each iteration of the outer loop uses the result of the previous iteration (`InG`) as input to calculate the next iteration (`G`). The recursive function `Loop` passes the result from one iteration to the next. While this may seem a bit complicated, it has the advantages of the declarative model. For example, it is straightforward to convert it into a concurrent algorithm for transitive closure using the model of Chapter 4. The concurrent algorithm can run efficiently on a parallel processor. We just add `thread . . . end` to parallelize the two outer loops as shown in Figure 6.13. This gives a parallel dataflow implementation of the algorithm. Synchronization is done through the tuples which initially contain unbound variables. The tuples behave like I-structures in a dataflow machine (see Section 4.9.5). It is an interesting exercise to draw a picture of an executing program, with data structures and threads.

Example executions

Let us calculate the transitive closure of the graph `[1#[2 3] 2#[1] 3#nil]`. This is the same graph we showed before in Figure 6.8 except that we use integers to represent the nodes. Here is how to use the declarative algorithms:

```
{Browse {DeclTrans [1#[2 3] 2#[1] 3#nil]}}
```

Here is how to use the stateful algorithm:

```
declare GM in
  {StateTrans GM={L2M [1#[2 3] 2#[1] 3#nil]}}
  {Browse {M2L GM}}
```

This is slightly more complicated because of the calls to `L2M` and `M2L`, which we use to give both the input and output as an adjacency list. All three algorithms give the result `[1#[1 2 3] 2#[1 2 3] 3#nil]`.

Discussion

Both the declarative and stateful algorithms are actually variations of the same conceptual algorithm, which is called the Floyd-Warshall algorithm. All three algorithms have an asymptotic running time of $O(n^3)$ for a graph of n nodes. So which algorithm is better? Let us explore different aspects of this question:

```

fun {DeclTrans2 GT}
  H={Width GT}
  fun {Loop K InG}
    if K=<H then
      G={MakeTuple g H} in
        thread
          for I in 1..H do
            thread
              G.I={MakeTuple g H}
              for J in 1..H do
                G.I.J=InG.I.J orelse
                  (InG.I.K andthen InG.K.J)
              end
            end
          end
        end
      {Loop K+1 G}
    else InG end
  end
in
  {Loop 1 GT}
end

```

Figure 6.13: Transitive closure (concurrent/parallel version)

- A first aspect is ease of understanding and reasoning. Perhaps surprisingly, the stateful algorithm has the simplest structure. It consists of three simple nested loops that update a matrix in a straightforward way. Both declarative algorithms have a more complex structure:
 - The first one takes an adjacency list and passes it through a sequence of stages in pipelined fashion. Each stage takes an input list and incrementally creates an output list.
 - The second one has a similar structure as the stateful algorithm, but creates a sequence of tuples in pipelined fashion.

Programming in the declarative model forces the algorithm to be structured as a pipeline, written with small, independent components. Programming in the stateful model encourages (but does not force) the algorithm to be structured as a monolithic block, which is harder to decompose. The stateful model gives more freedom in how to write the program. Depending on one's point of view, this can be a good or bad thing.

- A second aspect is performance: running time and memory use. Both algorithms asymptotically have the same running time and active memory sizes. We have measured the running times of both algorithms on several

large random graphs. Given a random graph of 200 nodes in which there is an edge between any node pair with probability p . For p greater than about 0.05, the first declarative algorithm takes about 10 seconds, the second about 12 seconds, and the stateful algorithm about 15 seconds. For p tending to 0, the first declarative algorithm tends towards 0 seconds and the other algorithms increase slightly, to 16 and 20 seconds, respectively.⁷ We conclude that the first declarative algorithm always has better performance than the two others. The adjacency list representation is better than the matrix representation when the graph is sparse.

Of course, the conclusions of this particular comparison are by no means definitive. We have chosen simple and clean versions of each style, but many variations are possible. For example, the first declarative algorithm can be modified to use a stateful `Union` operation. The stateful algorithm can be modified to stop looping when no more new edges are found. What then can we conclude from this comparison?

- Both the declarative and stateful models are reasonable for implementing transitive closure.
- The choice of representation (adjacency list or matrix) can be more important than the choice of computation model.
- Declarative programs tend to be less readable than stateful programs, because they must be written in pipelined fashion.
- Stateful programs tend to be more monolithic than declarative programs, because explicit state can be modified in any order.
- It can be easier to parallelize a declarative program, because there are fewer dependencies between its parts.

6.8.2 Word frequencies (with stateful dictionary)

In Section 3.7.3 we showed how to use dictionaries to count the number of different words in a text file. We compared the execution times of three versions of the word frequency counter, each one with a different implementation of dictionaries. The first two versions use declarative dictionaries (implemented lists and binary trees, respectively) and the third uses the built-in definition of dictionaries (implemented with state). The version using stateful dictionaries, shown in Figure 6.14, is slightly different from the one using declarative dictionaries, shown in Figure 3.29:

⁷All measurements using Mozart 1.1.0 under Red Hat Linux release 6.1 on a Dell Latitude CPx notebook computer with Pentium III processor at 500 MHz.


```

fun {WordChar C} ... end

fun {WordToAtom PW} ... end

fun {CharsToWords PW Cs} ... end

Put=Dictionary.put
CondGet=Dictionary.condGet

proc {IncWord D W}
  {Put D W {CondGet D W 0}+1}
end

proc {CountWords D Ws}
  case Ws
  of W|Wr then
    {IncWord D W}
    {CountWords D Wr}
  [] nil then skip
  end
end

fun {WordFreq Cs}
  D={NewDictionary}
in
  {CountWords D {CharsToWords nil Cs}}
  D
end

```

Figure 6.14: Word frequencies (with stateful dictionary)

- The stateful version needs to pass just one argument as input to each procedure that uses a dictionary.
- The declarative version has to use two arguments to these procedures: one for the input dictionary and one for the output dictionary. In Figure 3.29, the second output is realized by using functional notation.

The difference shows up in the operations `Put`, `IncWords`, `CountWords`, and `WordFreq`. For example, Figure 6.14 uses the stateful `{Put D LI X}`, which updates `D`. Figure 3.29 uses the declarative `{Put D1 LI X D2}`, which reads `D1` and returns a new dictionary `D2`.

6.8.3 Generating random numbers

A very useful primitive operation is a random number generator. It lets the computer “throw dice”, so to speak. How do we generate random numbers in a

computer? Here we give the main insights; see Knuth [101] for a deep discussion of the theory underlying random number generators and of the concept of randomness itself.

Different approaches

One could imagine the following ways to generate random numbers:

- A first technique would be to use unpredictable events in the computer itself, e.g., related to concurrency, as explained in the previous chapter. Alas, their unpredictability does not follow simple laws. For example, using the thread scheduler as a source of randomness will give some fluctuations, but they do not have a useful probability distribution. Furthermore, they are intimately linked with the computation in nonobvious ways, so even if their distribution was known, it would be dependent on the computation. So this is not a good source of random numbers.
- A second technique would be to rely on a source of true randomness. For example, electronic circuits generate *noise*, which is a completely unpredictable signal whose approximate probability distribution is known. The noise comes from the depths of the quantum world, so for all practical purposes it is truly random. But there are two problems. First, the probability distribution is not *exactly* known: it might vary slightly from one circuit to the next or with the ambient temperature. The first problem is not serious; there are ways to “normalize” the random numbers so that their distribution is a constant, known one. There is a second, more serious problem: the randomness cannot be reproduced except by storing the random numbers and replaying them. It might seem strange to ask for *reproducibility* from a source of randomness, but it is perfectly reasonable. For example, the randomness might be input to a simulator. We would like to vary some parameter in the simulator such that any variation in the simulator depends only on the parameter, and not on any variation in the random numbers. For this reason, computers are not usually connected to truly random sources.
- It might seem that we have carefully worked ourselves into a tight corner. We would like true randomness and we would like it to be reproducible. How can we resolve this dilemma? The solution is simple: we *calculate* the random numbers. How can this generate truly random numbers? The simple answer is, it cannot. But the numbers can appear random, *for all practical purposes*. They are called *pseudorandom* numbers. What does this mean? It is not simple to define. Roughly, the generated numbers should give the same behavior as truly random numbers, for the use we make of them.

The third solution, calculating random numbers, is the one that is almost always implemented. The question is, what algorithm do we use? Certainly not an algorithm chosen at random! Knuth [101] shows the pitfalls of this approach. It almost always gives bad results. We need an algorithm that has known good properties. We cannot *guarantee* that the random numbers will be good enough, but we can try to get what we can. For example, the generated random numbers should satisfy strong statistical properties, have the right distribution, and their period should be sufficiently long. The last point is worth expanding on: since a random number generator does a calculation with finite information, eventually it will repeat itself. Clearly, the period of repetition should be very long.

Uniformly distributed random numbers

A random number generator stores an internal state, with which it calculates the next random number and the next internal state. The state should be large enough to allow a long period. The random number is initialized with a number called its *seed*. Initializing it again with the same seed should give the same sequence of random numbers. If we do not want the same sequence, we can initialize it with information that will never be the same, such as the current date and time. Modern computers almost always have an operation to get the time information. Now we can define the abstract data type of a random number generator:

- `{NewRand ?Rand ?Init ?Max}` returns three references: a random number generator `Rand`, its initialization procedure `Init`, and its maximum value `Max`. Each generator has its own internal state. For best results, `Max` should be large. This allows the program to reduce the random numbers to the smaller domains it needs for its own purposes.
- `{Init Seed}` initializes the generator with integer seed `Seed`, that should be in the range `0, 1, ..., Max`. To give many possible sequences, `Max` should be large. Initialization can be done at any time.
- `X={Rand}` generates a new random number `X` and updates the internal state. `X` is an integer in the range `0, 1, ..., Max-1` and has a Uniform distribution, i.e., all integers have the same probability of appearing.

How do we calculate a new random number? It turns out that a good simple method is the *linear congruential* generator. If x is the internal state and s is the seed, then the internal state is updated as follows:

$$\begin{aligned}x_0 &= s \\x_n &= (ax_{n-1} + b) \bmod m\end{aligned}$$

The constants a , b , and m have to be carefully chosen so that the sequence x_0, x_1, x_2, \dots , has good properties. The internal state x_i is a uniformly distributed integer from 0 to $m - 1$. It is easy to implement this generator:

```

local
  A=333667
  B=213453321
  M=1000000000
in
  proc {NewRand ?Rand ?Init ?Max}
    X={NewCell 0}
  in
    proc {Init Seed} X:=Seed end
    fun {Rand} X:=(A*@X+B) mod M in @X end
    Max=M
  end
end

```

This is one of the simplest methods that has reasonably good behavior. More sophisticated methods are possible that are even better.

Using laziness instead of state

The linear congruential algorithm can be packaged in a completely different way, as a lazy function. To get the next random number, it suffices to read the next element of the stream. Here is the definition:

```

local
  A=333667
  B=213453321
  M=1000000000
in
  fun lazy {RandList S0}
    S1=(A*S0+B) mod M
  in
    S1|{RandList S1}
  end
end

```

Instead of using a cell, the state is stored in a recursive argument of `RandList`. Instead of calling `Rand` explicitly to get the next number, `RandList` is called implicitly when the next number is needed. Laziness acts as a kind of brake, making sure that the computation advances only as rapidly as its results are needed. A third difference is that higher-order programming is not needed, since each call to `RandList` generates a new sequence of random numbers.

Nonuniform distributions

A good technique to generate random numbers of any distribution is to start with a uniformly distributed random number. From this, we calculate a number with another distribution. Using this technique we explain how to generate Gaussian and Exponential distributions. We first define a new generator:

```
declare Rand Init Max in {NewRand Rand Init Max}
```

Now we define functions to generate a Uniform distribution from 0 to 1 and a Uniform integer distribution from A to B inclusive:

```
FMax={IntToFloat Max}
fun {Uniform}
  {IntToFloat {Rand}}/FMax
end

fun {UniformI A B}
  A+{FloatToInt {Floor {Uniform}*{IntToFloat B-A+1}}}
end
```

We will use `Uniform` to generate random variables with other distributions. First, let us generate random variables with an Exponential distribution. For this distribution, the probability that $X \leq x$ is $D(x) = 1 - e^{-\lambda x}$, where λ is a parameter called the *intensity*. Since $X \leq x$ iff $D(X) \leq D(x)$, it follows that the probability that $D(X) \leq D(x)$ is $D(x)$. Writing $y = D(x)$, it follows that the probability that $D(X) \leq y$ is y . Therefore $D(X)$ is uniformly distributed. Say $D(X) = U$ where U is a uniformly distributed random variable. Then we have $X = -\ln(1 - U)/\lambda$. This gives the following function:

```
fun {Exponential Lambda}
  ~{Log 1.0-{Uniform}}/Lambda
end
```

Now let us generate a Normal distribution with mean 0 and variance 1. This is also called a Gaussian distribution. We use the following technique. Given two variables U_1 and U_2 , uniformly distributed from 0 to 1. Let $R = \sqrt{-2 \ln U_1}$ and $\phi = 2\pi U_2$. Then $X_1 = R \cos \phi$ and $X_2 = R \sin \phi$ are independent variables with a Gaussian distribution. The proof of this fact is beyond the scope of the book; it can be found in [101]. This gives the following function:

```
TwoPi=4.0*{Float.acos 0.0}
fun {Gauss}
  {Sqrt ~2.0*{Log {Uniform}}}} * {Cos TwoPi*{Uniform}}
end
```

Since each call can give us *two* Gaussian variables, we can use a cell to remember one result for the next call:

```
local GaussCell={NewCell nil} in
  fun {Gauss}
    Prev={Exchange GaussCell $ nil}
  in
    if Prev\=nil then Prev
    else R Phi in
      R={Sqrt ~2.0*{Log {Uniform}}}}
      Phi=TwoPi*{Uniform}
      GaussCell:=R*{Cos Phi}
```

```

        R*{Sin Phi}
    end
end
end

```

Each call of `Gauss` calculates two independent Gaussian variables; we return one and store the other in a cell. The next call returns it without doing any calculation.

6.8.4 “Word of Mouth” simulation

Let us simulate how Web users “surf” on the Internet. To “surf” between Web sites means to successively load different Web sites. To keep our simulator simple, we will only look at one aspect of a Web site, namely its performance. This can be reasonable when surfing between Web *portals*, which each provide a large and similar set of services. Assume there are n Web sites with equal content and a total of m users. Each Web site has constant performance. Each user would like to get to the Web site with highest performance. But there is no global measure of performance; the only way a user can find out about performance is by asking other users. We say that information passes by “word of mouth”. This gives us the following simulation rules:

- Each site has a constant performance. Assume the constants are uniformly distributed.
- Each user knows which site it is on.
- Each site knows how many users are on it.
- Each user tries to step to a site with higher performance. The user asks a few randomly-picked users about the performance at their site. The user then goes to the site with highest performance. However, the performance information is not exact: it is perturbed by Gaussian noise.
- One round of the simulation consists of all users doing a single step.

With these rules, we might expect users eventually to swarm among the sites with highest performance. But is it really so? A simulation can give us answer.

Let us write a small simulation program. First, let us set up the global constants. We use the functions `Init`, `UniformI`, and `Gauss` defined in the previous section. There are n sites, m users, and we do t simulation rounds. We initialize the random number generator and write information to the file `wordofmouth.txt` during the simulation. We use the incremental write operations defined in the `File` module on the book’s Web site. With 10000 sites, 500000 users, and 200 rounds, this gives the following:

```

declare
N=10000 M=500000 T=200
{Init 0}
{File.writeOpen `wordofmouth.txt`}
proc {Out S}
    {File.write {Value.toVirtualString S 10 10}#"\\n"}
end

```

Next, we decide how to store the simulation information. We would like to store it in records or tuples, because they are easy to manipulate. But they cannot be modified. Therefore, we will store the simulation information in dictionaries. Dictionaries are very similar to records except that they can be changed dynamically (see Section 6.5.1). Each site picks its performance randomly. It has a dictionary giving its performance and the number of users on it. The following code creates the initial site information:

```

declare
Sites={MakeTuple sites N}
for I in 1..N do
    Sites.I={Record.toDictionary
        o(hits:0 performance:{IntToFloat {UniformI 1 80000}})}
end

```

Each user picks its site randomly. It has a dictionary giving its current site. It updates the Sites information. The following code creates the initial user information:

```

declare
Users={MakeTuple users M}
for I in 1..M do
    S={UniformI 1 N}
in
    Users.I={Record.toDictionary o(currentSite:S)}
    Sites.S.hits := Sites.S.hits + 1
end

```

Now that we have all the data structures, let us do one user step in the simulation. The function {UserStep I} does one step for user I, i.e., the user asks three other users for the performance of their sites, it calculates its new site, and then it updates all the site and user information.

```

proc {UserStep I}
    U = Users.I
    % Ask three users for their performance information
    L = {List.map [{UniformI 1 M} {UniformI 1 M} {UniformI 1 M}]}
    fun {$ X}
        (Users.X.currentSite) #
        Sites.(Users.X.currentSite).performance
        + {Gauss}*{IntToFloat N}
    end

```

```

    % Calculate the best site
    MS#MP = {List.foldL L
      fun {$ X1 X2} if X2.2>X1.2 then X2 else X1 end end
      U.currentSite #
      Sites.(U.currentSite).performance
      + {Abs {Gauss}*{IntToFloat N}}
    }
  in
    if MS\=U.currentSite then
      Sites.(U.currentSite).hits :=
        Sites.(U.currentSite).hits - 1
      U.currentSite := MS
      Sites.MS.hits := Sites.MS.hits + 1
    end
  end
end

```

Now we can do the whole simulation:

```

for J in 1..N do
  {Out {Record.adjoinAt {Dictionary.toRecord site Sites.J}
    name J}}
end
{Out endOfRound(time:0 nonZeroSites:N)}
for I in 1..T do
  X = {NewCell 0}
  in
    for U in 1..M do {UserStep U} end
    for J in 1..N do
      H=Sites.J.hits in
        if H\=0 then
          {Out {Record.adjoinAt
            {Dictionary.toRecord site Sites.J} name J}}
          X:=1+@X
        end
      end
    {Out endOfRound(time:I nonZeroSites:@X)}
  end
{File.writeClose}

```

To make the simulator self-contained, we put all the above code in one procedure with parameters N , M , T , and the output filename.

What is the result of the simulation? Will users cluster around the sites with highest performance, even though they have only a very narrow and inaccurate view of what is going on? Running the above simulation shows that the number of nonzero sites (with at least one user) decreases smoothly in inverse exponential fashion from 10000 initially to less than 100 after 83 rounds. Average performance of user sites increases from about 40000 (half of the maximum) to more than 75000 (within 6% of maximum) after just 10 rounds. So we can make a pre-

liminary conclusion that the best sites will quickly be found and the worst sites will quickly be abandoned, even by word-of-mouth propagation of very approximate information. Of course, our simulation has some simplifying assumptions. Feel free to change the assumptions and explore. For example, the assumption that a user can pick *any* three other users is unrealistic—it assumes that each user knows all the others. This makes convergence too fast. See the Exercises of this chapter for a more realistic assumption on user knowledge.

6.9 Advanced topics

6.9.1 Limitations of stateful programming

Stateful programming has some strong limitations due to its use of explicit state. Object-oriented programming is a special case of stateful programming, so it suffers from the same limitations.

The real world is parallel

The main limitation of the stateful model is that programs are sequential. In the real world, entities are both stateful and act in parallel. Sequential stateful programming does not model the parallel execution.

Sometimes this limitation is appropriate, e.g., when writing simulators where all events must be coordinated (stepping from one global state to another in a controlled way). In other cases, e.g., when interacting with the real world, the limitation is an obstacle. To remove the limitation, the model needs to have both state and concurrency. We have seen one simple way to achieve this in Chapter 5. Another way is given in Chapter 8. As Section 4.7.6 explains, concurrency in the model can model parallelism in the real world.

The real world is distributed

Explicit state is hard to use well in a distributed system. Chapter 11 explains this limitation in depth. Here we give just the main points. In a distributed system, the store is partitioned into separate parts. Within one part, the store behaves efficiently as we have seen. Between parts, communication is many orders of magnitude more expensive. The parts coordinate with one another to maintain the desired level of global consistency. For cells this can be expensive because cell contents can change at any time in any part. The programmer has to decide on both the level of consistency and the coordination algorithm used. This makes it tricky to do distributed programming with state.

The declarative model and its extension to concurrent message passing in Chapter 5 are much easier to use. As Chapter 5 explains, a system can be decomposed into independent components that communicate with messages. This

fits very well with the partitioned store of a distributed system. When programming a distributed system, we recommend to use the message-passing model whenever possible for coordinating the parts. Chapter 11 explains how to program a distributed system and when to use the different computation models in a distributed setting.

6.9.2 Memory management and external references

As explained in Section 2.4.7, garbage collection is a technique for automatic memory management that recovers memory for all entities inside the computation model that no longer take part in the computation. This is not good enough for entities *outside* the computation model. Such entities exist because there is a world outside of the computation model, which interacts with it. How can we do automatic memory management for them? There are two cases:

- From inside the computation model, there is a reference to an entity outside it. We call such a reference a *resource pointer*. Here are some examples:
 - A file descriptor, which points to a data structure held by the operating system. When the file descriptor is no longer referenced, we would like to close the file.
 - A handle to access an external database. When the handle is no longer referenced, we would like to close the connection to the database.
 - A pointer to a block of memory allocated through the Mozart C++ interface. When the memory is no longer referenced, we would like it to be freed.
- From the external world, there is a reference to inside the computation model. We call such a reference a *ticket*. Tickets are used in distributed programming as a means to connect processes together (see Chapter 11).

In the second case, there is no safe way in general to recover memory. By *safe* we mean not to release memory as long as external references exist. The external world is so big that the computation model cannot know whether any reference still exists or not. One pragmatic solution is to add the language entity to the root set for a limited period of time. This is known as a *time-lease mechanism*. The time period can be renewed when the language entity is accessed. If the time period expires without a renewal, we assume that there are no more external references. The application has to be designed to handle the rare case when this assumption is wrong.

In the first case, there is a simple solution based on parameterizing the garbage collector. This solution, called *finalization*, gives the ability to perform a user-defined action when a language entity has become unreachable. This is implemented by the System module `Finalize`. We first explain how the module works. We then give some examples of how it is used.

Finalization

Finalization is supported by the `Finalize` module. The design of this module is inspired by the *guardian* concept of [51]. `Finalize` has the following two operations:

- `{Finalize.register X P}` registers a reference `X` and a procedure `P`. When `X` becomes otherwise unreachable (otherwise than through finalization), `{P X}` is eventually executed in its own thread. During this execution, `X` is reachable again until its reference is no longer accessible.
- `{Finalize.everyGC P}` registers a procedure `P` to be invoked eventually after every garbage collection.

In both of these operations, you cannot rely on how soon after the garbage collection the procedure `P` will be invoked. It is in principle possible that the call may only be scheduled several garbage collections late if the system has very many live threads and generates garbage at a high rate.

There is no limitation on what the procedure `P` is allowed to do. This is because `P` is not executed *during* garbage collection, when the system's internal data structures can be temporarily inconsistent, but is scheduled for execution *after* garbage collection. `P` can even reference `X` and itself call `Finalize`.

An interesting example is the `everyGC` operation itself, which is defined in terms of `register`:

```

proc {EveryGC P}
  proc {DO _} {P} {Finalize.register DO DO} end
in
  {Finalize.register DO DO}
end

```

This creates a procedure `DO` and registers it using itself as its own handler. When `EveryGC` exits, the reference to `DO` is lost. This means that `DO` will be invoked after the next garbage collection. When invoked, it calls `P` and registers itself again.

Laziness and external resources

To make lazy evaluation practical for external resources like files, we need to use finalization to release the external resources when they are no longer needed. For example, in Section 4.5.5 we defined a function `ReadListLazy` that reads a file lazily. This function closes the file after it is completely read. But this is not good enough: even if only part of the file is needed, the file should also be closed. We can implement this with finalization. We extend the definition of `ReadListLazy` to close the file when it becomes inaccessible:

```

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
  {Finalize.register F proc {$ F} {File.readClose} end}
  {ReadNext}
end

```

This requires just one call to `Finalize`.

6.10 Exercises

1. **What is state.** Section 6.1 defines the function `SumList`, which has a state encoded as the successive values of two arguments at recursive calls. For this exercise, rewrite `SumList` so that the state is no longer encoded in arguments, but by cells.
2. **Emulating state with concurrency.** This exercise explores whether concurrency can be used to obtain explicit state.
 - (a) First use concurrency to create an updatable container. We create a thread that uses a recursive procedure to read a stream. The stream has two possible commands: `access(X)`, which binds `X` to the container's current content, and `assign(X)`, which assigns `X` as the new content. Here is how it is done:

```

fun {MakeState Init}
  proc {Loop S V}
    case S of access(X) | S2 then
      X=V {Loop S2 V}
    [] assign(X) | S2 then
      {Loop S2 X}
    else skip end
  end
  S
in
  thread {Loop S Init} end
  S
end

S={MakeState 0}

```

The call `{MakeState Init}` creates a new container with initial content `Init`. We use the container by putting commands on the stream.

For example, here is a sequence of three commands for the container `S`:

```
declare S1 X Y in
  S=access(X) | assign(3) | access(Y) | S1
```

This binds `x` to 0 (the initial content), puts 3 in the container, and then binds `y` to 3.

- (b) Now rewrite `SumList` to use this container to count the number of calls. Can this container be encapsulated, i.e., can it be added without changing the arguments of `SumList`? Why or why not? What happens when we try to add the function `SumCount` like in Section 6.1.2?
3. **Implementing ports.** In Chapter 5 we introduced the concept of *port*, which is a simple communication channel. Ports have the operations `{NewPort S P}`, which returns a port `P` with stream `S`, and `{Send P x}`, which sends message `x` on port `P`. From these operations, it is clear that ports are a stateful unbundled ADT. For this exercise, implement ports in terms of cells, using the techniques of Section 6.4.
4. **Explicit state and security.** Section 6.4 gives four ways to construct secure ADTs. From these constructions, it seems that the ability to make ADTs secure is a consequence of using one or both of the following concepts: procedure values (which provide hiding through lexical scoping) and name values (which are unforgeable and unguessable). In particular, explicit state seems to have no role with respect to security. For this exercise, think carefully about this assertion. Is it true? Why or why not?
5. **Declarative objects and identity.** Section 6.4.2 shows how to build a declarative object, which combines value and operations in a secure way. However, the implementation given misses one aspect of objects, namely their identity. That is, an object should keep the same identity after state changes. For this exercise, extend the declarative objects of Section 6.4.2 to have an identity.
6. **Revocable capabilities.** Section 6.4.3 defines the three-argument procedure `Revocable`, which takes a capability and uses explicit state to create two things: a revocable version of that capability and a revoker. For `Revocable`, the capability is represented as a one-argument procedure and the revoker is a zero-argument procedure. For this exercise, write a version of `Revocable` that is a one-argument procedure and where the revoker is also a one-argument procedure. This allows `Revocable` to be used recursively on all capabilities including revokers and itself. For example, the ability to revoke a capability can then be made revocable.
7. **Abstractions and memory management.** Consider the following ADT which allows to collect information together into a list. The ADT has three

operations. The call `C={NewCollector}` creates a new collector `C`. The call `{Collect C X}` adds `X` to `C`'s collection. The call `L={EndCollect}` returns the final list containing all collected items in the order they were collected. Here are two ways to implement collectors that we will compare:

- `C` is a cell that contains a pair `H|T`, where `H` is the head of the collected list and `T` is its unbound tail. `Collect` is implemented as:

```
proc {Collect C X}
  H T in
    {Exchange C H|(X|T) H|T}
  end
```

Implement the `NewCollector` and `EndCollect` operations with this representation.

- `C` is a pair `H|T`, where `H` is the head of the collected list and `T` is a cell that contains its unbound tail. `Collect` is implemented as:

```
proc {Collect C X}
  T in
    {Exchange C.2 X|T T}
  end
```

Implement the `NewCollector` and `EndCollect` operations with this representation.

- We compare the two implementations with respect to memory management. Use the table of Section 3.5.2 to calculate how many words of memory are allocated by each version of `Collect`. How many of these words immediately become inactive in each version? What does this imply for garbage collection? Which version is best?

This example is taken from the Mozart system. Collection in the **for** loop was originally implemented with one version. It was eventually replaced by the other. (Note that both versions work correctly in a concurrent setting, i.e., if `Collect` is called from multiple threads.)

8. **Call by name.** Section 6.4.4 shows how to code call by name in the stateful computation model. For this exercise, consider the following example taken from [56]:

```
procedure swap(callbyname x,y:integer);
var t:integer;
begin
  t:=x; x:=y; y:=t
end;

var a:array [1..10] of integer;
```

```

var i:integer;

i:=1; a[1]:=2; a[2]=1;
swap(i, a[i]);
writeln(a[1], a[2]);

```

This example shows a curious behavior of call by name. Running the example does *not* swap `i` and `a[i]`, as one might expect. This shows an undesirable interaction between destructive assignment and the delayed evaluation of an argument.

- Explain the behavior of this example using your understanding of call by name.
- Code the example in the stateful computation model. Use the following encoding of `array[1..10]`:

```

A={MakeTuple array 10}
for J in 1..10 do A.J={NewCell 0} end

```

That is, code the array as a tuple of cells.

- Explain the behavior again in terms of your coding.
9. **Call by need.** With call by name, the argument is evaluated again each time it is needed.
 - For this exercise, redo the swap example of the previous exercise with call by need instead of call by name. Does the counterintuitive behavior still occur? If not, can similar problems still occur with call by need by changing the definition of `swap`?
 - In the code that implements call by need, `Sqr` will always call `A`. This is fine for `Sqr`, since we can see by inspection that the result is needed three times. But what if the need cannot be determined by inspection? We do not want to call `A` unnecessarily. One possibility is to use lazy functions. Modify the coding of call by need given in Section 6.4.4 so that it uses laziness to call `A` only when needed, even if that need cannot be determined by inspection. `A` should be called at most once.
 10. **Evaluating indexed collections.** Section 6.5.1 presents four indexed collection types, namely tuples, records, arrays, and dictionaries, with different performance/expressiveness trade-offs. For this exercise, compare these four types in various usage scenarios. Evaluate their relative performance and usefulness.
 11. **Extensible arrays.** The extensible array of Section 6.5 only extends the array *upwards*. For this exercise, modify the extensible array so it extends the array in both directions.

12. **Generalized dictionaries.** The built-in dictionary type only works for literal keys, i.e., numbers, atoms, or names. For this exercise, implement a dictionary that can use any value as a key. One possible solution uses the fact that the `==` operation can compare any values. Using this operation, the dictionary could store entries as an association list, which is a list of pairs `Key#Value`, and do simple linear search.
13. **Loops and invariant assertions.** Use the method of invariant assertions to show that the proof rules for the **while** and **for** loops given in Section 6.6.4 are correct.
14. **The break statement.** A *block* is a set of statements with a well-defined entry point and exit point. Many modern imperative programming languages, such as Java and C++, are based on the concept of block. These languages allow defining nested blocks and provide an operation to jump immediately from within a block to the block's exit point. This operation is called **break**. For this exercise, define a block construct with a break operation that can be called as follows:

```
{Block proc {$ Break} <stmt> end}
```

This should have exactly the same behavior as executing `<stmt>`, except that executing `{Break}` inside `<stmt>` should immediately exit the block. Your solution should work correctly for nested blocks and exceptions raised within blocks. If `<stmt>` creates threads, then these should not be affected by the break operation. Hint: use the exception handling mechanism.

15. **“Small World” simulation.** The “Word of Mouth” simulation of Section 6.8.4 makes some strong simplifying assumptions. For example, the simulation assumes that each user can choose any three users at random to ask them about their performance. This is much too strong an assumption. The problem is that the choice ranges over *all* users. This gives each user a potentially unbounded amount of knowledge. In actuality, each user has bounded knowledge: a small network of acquaintances that changes but slowly. Each user asks only members of his network of acquaintances. Rewrite the simulation program to take this assumption into account. This can make convergence much slower. With this assumption, the simulation is called a “Small World” simulation [203].
16. **Performance effects in “Word of Mouth” simulation.** The “Word of Mouth” simulation of Section 6.8.4 assumes that site performance is constant. A better way to take performance into account is to assume that it is constant up to a given threshold number of users, which is fixed for each site. Beyond this threshold, performance goes down in inverse proportion to the number of users. This is based on the premise that for small numbers of users, Internet performance is the bottleneck, and for large numbers of users, site performance is the bottleneck.

17. **Word frequency application.** Section 6.8.2 gives a version of the word frequency algorithm that uses stateful dictionaries. Rewrite the word frequency application of Section 3.9.4 to use the stateful version.

Chapter 7

Object-Oriented Programming

“The fruit is too well known to need any description of its external characteristics.”

– From entry “Apple”, *Encyclopaedia Britannica* (11th edition)

This chapter introduces a particularly useful way of structuring stateful programs called *object-oriented programming*. It introduces one new concept over the last chapter, namely *inheritance*, which allows to define ADTs in incremental fashion. However, the computation model is the same stateful model as in the previous chapter. We can loosely define object-oriented programming as programming with encapsulation, explicit state, and inheritance. It is often supported by a linguistic abstraction, the concept of *class*, but it does not have to be. Object-oriented programs can be written in almost any language.

From a historical viewpoint, the introduction of object-oriented programming made two major contributions to the discipline of programming. First, it made clear that encapsulation is essential. Programs should be organized as collections of ADTs. This was first clearly stated in the classic article on “information hiding” [142], reprinted in [144]. Each module, component, or object has a “secret” known only to itself. Second, it showed the importance of building ADTs incrementally, using inheritance. This avoids duplicated code.

Object-oriented programming is one of the most successful and pervasive areas in informatics. From its timid beginnings in the 1960’s it has invaded every area of informatics, both in scientific research and technology development. The first object-oriented language was Simula 67, developed in 1967 as a descendant of Algol 60 [130, 137, 152]. Simula 67 was much ahead of its time and had little immediate influence. Much more influential in making object-oriented programming popular was Smalltalk-80, released in 1980 as the result of research done in the 1970’s [60]. The currently most popular programming languages, Java and C++, are object-oriented [186, 184]. The most popular “language-independent” design aids, the Unified Modeling Language (UML) and Design Patterns, both implicitly assume that the underlying language is object-oriented [58, 159]. With

all this exposure, one might feel that object-oriented programming is well understood (see the chapter quote). Yet, this is far from being the case.

Structure of the chapter

The purpose of this chapter is not to cover all of object-oriented programming in 100 pages or less. This is impossible. Instead, we give an introduction that emphasizes areas where other programming books are weak: the relationship with other computation models, the precise semantics, and the possibilities of dynamic typing. The chapter is structured as follows:

- **Motivations** (Section 7.1). We give the principal motivation for object-oriented programming, namely to support inheritance, and how its features relate to this.
- **An object-oriented computation model** (Sections 7.2 and 7.3). We define an object system that takes advantage of dynamic typing to combine simplicity and flexibility. This allows us to explore better the limits of the object-oriented abstraction and situate existing languages within them. We single out three areas: controlling encapsulation, single and multiple inheritance, and higher-order programming techniques. We give the object system syntactic and implementation support to make it easier to use and more efficient.
- **Programming with inheritance** (Section 7.4). We explain the basic principles and techniques for using inheritance to construct object-oriented programs. We illustrate them with realistic example programs. We give pointers into the literature on object-oriented design.
- **Relation to other computation models** (Section 7.5). From the viewpoint of multiple computation models, we show how and when to use and not use object-oriented programming. We relate it to component-based programming, object-based programming, and higher-order programming. We give additional design techniques that become possible when it is used together with other models. We explain the pros and cons of the oft-repeated principle stating that every language entity should be an object. This principle has guided the design of several major object-oriented languages, but is often misunderstood.
- **Implementing the object system** (Section 7.6). We give a simple and precise semantics of our object system, by implementing it in terms of the stateful computation model. Because the implementation uses a computation model with a precise semantics, we can consider it as a semantic definition.

- **The Java language** (Section 7.7). We give an overview of the sequential part of Java, a popular object-oriented programming language. We show how the concepts of Java fit in the object system of the chapter.
- **Active objects** (Section 7.8). An active object extends a port object of Chapter 5 by using a class to define its behavior. This combines the abilities of object-oriented programming with message-passing concurrency.

After reading this chapter, you will have a better view of what object-oriented programming is about, how to situate it among other computation models, and how to use the expressiveness it offers.

Object-Oriented Software Construction

For more information on object-oriented programming techniques and principles, we recommend the book *Object-Oriented Software Construction, Second Edition*, by Bertrand Meyer [122]. This book is especially interesting for its detailed discussion of inheritance, including multiple inheritance.

7.1 Motivations

7.1.1 Inheritance

As we saw in the previous chapter, stateful abstract data types are a very useful concept for organizing a program. In fact, a program can be built in a hierarchical structure as ADTs that depend on other ADTs. This is the idea of component-based programming.

Object-oriented programming takes this idea one step further. It is based on the observation that components frequently have much in common. Take the example of sequences. There are many different ADTs that are “sequence-like”. Sometimes we want them to behave like stacks (adding and deleting at the same end). Sometimes we want them to behave like queues (adding and deleting at opposite ends). And so forth, with dozens of possibilities. All of these sequences share the basic, linear-order property of the concept of sequence. How can we implement them without duplicating the common parts?

Object-oriented programming answers this question by introducing the additional concept of *inheritance*. An ADT can be defined to “inherit” from other ADTs, that is, to have substantially the same functionality as the others, with possibly some modifications and extensions. Only the *differences* between the ADT and its ancestors have to be specified. Such an incremental definition of an ADT is called a *class*.

Stateful model with inheritance

Inheritance is the essential difference between object-oriented programming and most other kinds of stateful programming. It is important to emphasize that inheritance is a *programming technique*; the underlying computation model of object-oriented programming is simply the stateful model (or the shared-state concurrent model, for concurrent object-oriented programming). Object-oriented languages provide linguistic support for inheritance by adding classes as a linguistic abstraction.

Caveats

It turns out that inheritance is a very rich concept that can be rather tricky. There are many ways that an ADT can be built by modifying other ADTs. The primary approach used in object-oriented programming is *syntactic*: a new ADT is defined by doing simple syntactic manipulations of an existing ADT. Because the resulting changes in semantics are not always easy to infer, these manipulations must be done with great care.

The component approach to building systems is much simpler. A component groups together any set of entities and treats them as a unit from the viewpoint of use dependency. A component is built from subcomponents, respecting their specifications.

Potential

Despite the difficulties of using inheritance, it has a great potential: it increases the possibilities of *factoring* an application, i.e., to make sure that each abstraction is implemented just once. Having more than one implementation of an abstraction does not just make the program longer. It is an invitation to disaster: if one implementation is changed, then the others must also be changed. What's more, the different implementations are usually slightly different, which makes nonobvious the relationships among all the changes. This "code duplication" of an abstraction is one of the biggest sources of errors. Inheritance has the potential to remove this duplication.

The potential to factor an application is a two-edged sword. It comes at the price of "spreading out" an ADT's implementation over large parts of the program. The implementation of an ADT does not exist in one place; all the ADTs that are part of it have to be considered together. Even stronger, part of the implementation may exist only as compiled code, with no access to the source code.

Early on, it was believed that inheritance would solve the problem of software reuse. That is, it would make it easier to build libraries that can be distributed to third parties, for use in other applications. This has not worked out in practice. The failure of inheritance as a reuse technique is clear from the success of

other techniques such as components, frameworks, and design patterns. Inheritance remains most useful within a single application or closely-related family of applications.

Inheritance is not an unmixed blessing, but it takes its place next to higher-order programming as one of the most important techniques for structuring a program.

7.1.2 Encapsulated state and inheritance

The combination of encapsulating explicit state and inheritance has led to the field of *object-oriented programming*, which is presented in this chapter. This field has developed a rich theory and practice on how to write stateful programs with inheritance. Unfortunately, this theory tends to consider everything as being an object and to mix the notions of state and encapsulation. The advantages to be gained by considering other entities than objects and by using encapsulation without state are often ignored. Chapters 3 and 4 explain well how to use these two ideas. The present chapter follows the object-oriented philosophy and emphasizes how to build ADTs with both explicit state and inheritance.

Most object-oriented programming languages consider that ADTs should have explicit state by default. For example, Smalltalk, C++, and Java all consider variables to be stateful, i.e., mutable, by default. In Java it is possible to make variables immutable by declaring them as `final`, but it is not the default. This goes against the rule of thumb given in Section 4.7.6, and in our view it is a mistake. Explicit state is a complex notion which should not be the first one that students are taught. There are simpler ways to program, e.g., using variable identifiers to refer to values or dataflow variables. These simpler ways should be considered first before moving to explicit state.

7.1.3 Objects and classes

An *object* is an entity that encapsulates a state so that it can only be accessed in a controlled way from outside the object. The access is provided by means of *methods*, which are procedures that are accessible from the outside and that can directly access the internal state. The only way to modify the state is by calling the methods. This means that the object can guarantee that the state always satisfies some invariant property.

A *class* is an entity that specifies an object in an incremental way, by defining the classes that the object inherits from (its *direct ancestors*) and defining how the class is different from the direct ancestors. Most modern languages support classes as a linguistic abstraction. We will do the same in this chapter. To make the concepts precise we will add a simple yet powerful **class** construct.

This chapter only talks about objects that are used sequentially, i.e., that are used in a single thread. Chapter 8 explains how to use objects in a concurrent