5 Programming Language

We have been using a very simple programming language consisting of only ok , assignment, **if then else** , dependent (sequential) composition, and refined specifications. In this chapter we enrich our repertoire by considering some of the notations found in some popular languages. We will not consider concurrency (independent composition) and interaction (input and output) just yet; they get their own chapters later.

5.0 Scope

5.0.0 Variable Declaration

The ability to declare a new state variable within a local scope is so useful that it is provided by every decent programming language. A declaration may look something like this:

$$\mathbf{var}\ x\colon T$$

where x is the variable being declared, and T , called the type, indicates what values x can be assigned. A variable declaration applies to what follows it, according to the precedence table on the final page of the book. In program theory, it is essential that each of our notations apply to all specifications, not just to programs. That way we can introduce a local variable as part of the programming process, before its scope is refined.

We can express a variable declaration together with the specification to which it applies as a boolean expression in the initial and final state.

$$\mathbf{var}\ x\colon T\cdot P \quad = \quad \exists x, x'\colon T\cdot P$$

Specification P is an expression in the initial and final values of all nonlocal (already declared) variables plus the newly declared local variable. Specification $\mathbf{var}\ x\colon T\cdot P$ is an expression in the nonlocal variables only. For a variable declaration to be implementable, its type must be nonempty. As a simple example, suppose the nonlocal variables are integer variables y and z . Then

$$\begin{aligned} & \mathbf{var}\ x\colon int\cdot\ x:= 2.\ y:= x+z \\ = \quad & \exists x, x'\colon int\cdot\ x'=2\ \land\ y' = 2+z\ \land\ z'=z \\ = \quad & y' = 2+z\ \land\ z'=z \end{aligned}$$

According to our definition of variable declaration, the initial value of the local variable is an arbitrary value of its type.

$$\begin{aligned} & \mathbf{var}\ x\colon int\cdot\ y:= x \\ = \quad & \exists x, x'\colon int\cdot\ x'=x \land y'=x \land z'=z \\ = \quad & z'=z \end{aligned}$$

which says that z is unchanged. Variable x is not mentioned because it is a local variable, and variable y is not mentioned because its final value is unknown. However

$$\begin{aligned} & \mathbf{var}\ x\colon int\cdot\ y:= x-x \\ = \quad & y'=0 \land z'=z \end{aligned}$$

In some languages, a newly declared variable has a special value called “the undefined value” which cannot participate in any expressions. To write such declarations as boolean expressions, we introduce the expression $undefined$ but we do not give any axioms about it, so nothing can be proven about it. Then

$$\mathbf{var}\ x\colon T\cdot P \quad = \quad \exists x\colon undefined\cdot \exists x'\colon T, undefined\cdot P$$

For this kind of variable declaration, it is not necessary for the type to be nonempty.

An initializing assignment is easily defined in the same way.

$$\textbf{var}\ x: T := e \cdot\ P \quad = \quad \exists x: e \cdot \exists x': T \cdot P$$

assuming e is of type T.

If we are accounting for space usage, a variable declaration should be accompanied by an increase to the space variable s at the start of the scope of the declaration, and a corresponding decrease to s at the end of the scope.

As in many programming languages, we can declare several variables in one declaration. For example,

$$\textbf{var}\ x, y, z: T \cdot P \quad = \quad \exists x, x', y, y', z, z': T \cdot P$$

End of Variable Declaration

It is a service to the world to make variable declarations as local as possible. That way, the state space outside the local scope is not polluted with unwanted variables. Inside the local scope, there are all the nonlocal variables plus the local ones; there are more variables to keep track of locally.

5.0.1 Variable Suspension

We may wish, temporarily, to narrow our focus to a part of the state space. If the part is x and y, we indicate this with the notation

$$\textbf{frame}\ x, y$$

It applies to what follows it, according to the precedence table on the final page of the book, just like **var** . The **frame** notation is the formal way of saying "and all other variables (even the ones we cannot say because they are covered by local declarations) are unchanged". This is similar to the "import" statement of some languages, though not identical. If the state variables not included in the frame are w and z, then

$$\textbf{frame}\ x, y \cdot P \quad = \quad P \land w'=w \land z'=z$$

Within P the state variables are x and y. It allows P to refer to w and z, but only as local constants (mathematical variables, not state variables; there is no w' and no z'). Time and space variables are implicitly assumed to be in all frames, even though they may not be listed explicitly.

The definitions of ok and assignment using state variables

$$ok \quad = \quad x'=x \land y'=y \land ...$$
$$x:= e \quad = \quad x'=e \land y'=y \land ...$$

were partly informal, using three dots to say "and other conjuncts for other state variables". If we had defined **frame** first, we could have defined them formally as follows:

$$ok \quad = \quad \textbf{frame} \cdot \top$$
$$x:= e \quad = \quad \textbf{frame}\ x \cdot x'=e$$

End of Variable Suspension

We specified the list summation problem in the previous chapter as $s' = \Sigma L$. We took s to be a state variable, and L to be a constant. We might have preferred the specification $s:= \Sigma L$ saying that s has the right final value and that all other variables are unchanged, but our solution included a variable n which began at 0 and ended at $\#L$. We now have the formal notations needed.

$$s:= \Sigma L \quad = \quad \textbf{frame}\ s \cdot\ \textbf{var}\ n: nat \cdot\ s' = \Sigma L$$

First we reduce the state space to s ; if L was a state variable, it is now a constant. Next we introduce local variable n. Then we proceed as before.

End of Scope

5.1 Data Structures

5.1.0 Array

In most popular programming languages there is the notion of subscripted variable, or indexed variable, usually called an array. Each element of an array is a variable. Element 2 of array A can be assigned the value 3 by a notation such as

$$A(2):= 3$$

Perhaps the brackets are square; let us dispense with the brackets. We can write an array element assignment as a boolean expression in the initial and final state as follows. Let A be an array name, let i be any expression of the index type, and let e be any expression of the element type. Then

$$Ai:= e \quad = \quad A'i=e \ \land \ (\forall j\cdot j\neq i \Rightarrow A'j=Aj) \ \land \ x'=x \ \land \ y'=y \ \land \ ...$$

This says that after the assignment, element i of A equals e, all other elements of A are unchanged, and all other variables are unchanged. If you are unsure of the placement of the primes, consider the example

$$\begin{aligned} & A(A2):= 3 \\ = \quad & A'(A2) = 3 \ \land \ (\forall j\cdot j\neq A2 \Rightarrow A'j=Aj) \ \land \ x'=x \ \land \ y'=y \ \land \ ... \end{aligned}$$

The Substitution Law

$$x:= e.\ P \quad = \quad (\text{for } x \text{ substitute } e \text{ in } P)$$

is very useful, but unfortunately it does not work for array element assignment. For example,

$$A2:= 3.\ i:= 2.\ Ai:= 4.\ Ai=A2$$

should equal $\top$, because $i=2$ just before the final boolean expression, and $A2=A2$ certainly equals $\top$. If we try to apply the Substitution Law, we get

$$\begin{aligned} & A2:= 3.\ i:= 2.\ Ai:= 4.\ Ai=A2 && \text{invalid use of substitution law} \\ = \quad & A2:= 3.\ i:= 2.\ 4=A2 && \text{valid use of substitution law} \\ = \quad & A2:= 3.\ 4=A2 && \text{invalid use of substitution law} \\ = \quad & 4=3 \\ = \quad & \bot \end{aligned}$$

Here is a second example of the failure of the Substitution Law for array elements.

$$A2:= 2.\ \ A(A2):= 3.\ \ A2=2$$

This should equal $\bot$ because $A2=3$ just before the final boolean expression. But the Substitution Law says

$$\begin{aligned} & A2:= 2.\ \ A(A2):= 3.\ \ A2=2 && \text{invalid use of substitution law} \\ = \quad & A2:= 2.\ \ A2=2 && \text{invalid use of substitution law} \\ = \quad & 2=2 \\ = \quad & \top \end{aligned}$$

The Substitution Law works only when the assignment has a simple name to the left of $:=$. Fortunately we can always rewrite an array element assignment in that form.

$$\begin{aligned} & Ai:= e \\ = \quad & A'i=e \ \land \ (\forall j\cdot j\neq i \Rightarrow A'j=Aj) \ \land \ x'=x \ \land \ y'=y \ \land \ ... \\ = \quad & A' = i\rightarrow e \mid A \ \land \ x'=x \ \land \ y'=y \ \land \ ... \\ = \quad & A:= i\rightarrow e \mid A \end{aligned}$$

Let us look again at the examples for which the Substitution Law did not work, this time using the notation $A:= i\to e \mid A$.

$$
\begin{aligned}
& A:= 2\to 3 \mid A.\ \ i:= 2.\ \ A:= i\to 4 \mid A.\ \ Ai = A2 \\
= \quad & A:= 2\to 3 \mid A.\ \ i:= 2.\ \ (i\to 4 \mid A)i = (i\to 4 \mid A)2 \\
= \quad & A:= 2\to 3 \mid A.\ \ (2\to 4 \mid A)2 = (2\to 4 \mid A)2 \\
= \quad & A:= 2\to 3 \mid A.\ \ \top \\
= \quad & \top
\end{aligned}
$$

$$
\begin{aligned}
& A:= 2\to 2 \mid A.\ \ A:= A2\to 3 \mid A.\ \ A2=2 \\
= \quad & A:= 2\to 2 \mid A.\ \ (A2\to 3 \mid A)2 = 2 \\
= \quad & ((2\to 2 \mid A)2\to 3 \mid 2\to 2 \mid A)\ 2 = 2 \\
= \quad & (2\to 3 \mid 2\to 2 \mid A)\ 2 = 2 \\
= \quad & 3 = 2 \\
= \quad & \bot
\end{aligned}
$$

The only thing to remember about array element assignment is this: change $Ai:= e$ to $A:= i\to e \mid A$ before applying any programming theory. A two-dimensional array element assignment $Aij:= e$ must be changed to $A:= (i; j)\to e \mid A$, and similarly for more dimensions.

End of Array

5.1.1 Record

Without inventing anything new, we can already build records, also known as structures, similar to those found in several languages. Let us define *person* as follows.

$$
\begin{aligned}
person \ = \ & \text{"name"} \to text \\
\mid \ & \text{"age"} \to nat
\end{aligned}
$$

We declare

var p: *person*

and assign p as follows.

p:= "name" → "Josh" | "age" → 17

In languages with records (or structures), a component (or field) is assigned the same way we make an array element assignment. For example,

p "age":= 18

Just as for array element assignment, the Substitution Law does not work for record components. And the solution is also the same; just rewrite it like this:

p:= "age" → 18 | p

No new theory is needed for records.

End of Record

End of Data Structures

5.2 Control Structures

5.2.0 While Loop

The **while**-loop of several languages has a syntax similar to

while b **do** P

where b is boolean and P is a specification. To execute it, evaluate b , and if its value is $\bot$ then you're done, but if its value is $\top$ then execute P and start over. We do not define the **while**-loop as a specification the way we have defined previous programming notations. Instead, if W is an implementable specification, we consider

$W \Leftarrow$ **while** b **do** P

to be an abbreviation of

$W \Leftarrow$ **if** b **then** $(P.\ W)$ **else** ok

For example, to prove

$s' = s + \Sigma L[n;..\#L] \ \land\ t' = t + \#L - n \ \Leftarrow$
while $n \neq \#L$ **do** $(s:= s + Ln.\ n:= n+1.\ t:= t+1)$

prove instead

$s' = s + \Sigma L[n;..\#L] \ \land\ t' = t + \#L - n \ \Leftarrow$
if $n \neq \#L$ **then** $(s:= s + Ln.\ n:= n+1.\ t:= t+1.\ s' = s + \Sigma L[n;..\#L] \ \land\ t' = t+\#L-n)$
else ok

During programming, we may happen to refine a specification W by **if** b **then** $(P.\ W)$ **else** ok . If so, we may abbreviate the refinement using a while-loop. This is particularly valuable when the implementation of call is poor, and does not use a branch instruction in this situation.

This account of **while**-loops is adequate for practical purposes: it tells us how we can use them in programming. But it does not allow us to prove as much as we might like; for example, we cannot prove

while b **do** P $=$ **if** b **then** $(P.$ **while** b **do** $P)$ **else** ok

A different account of **while**-loops is given in Chapter 6.

Exercise 265: Consider the following program in natural variables x and y .

while $\neg\ x=y=0$ **do**
if $y>0$ **then** $y:= y-1$
else $(x:= x-1.$ **var** $n: nat\cdot\ y:= n)$

This loop decreases y until it is 0 ; then it decreases x by 1 and assigns an arbitrary natural number to y ; then again it decreases y until it is 0 ; and again it decreases x by 1 and assigns an arbitrary natural number to y ; and so on until both x and y are 0 . The problem is to find a time bound. So we introduce time variable t , and rewrite the loop in refinement form.

$P \Leftarrow$ **if** $x=y=0$ **then** ok
else if $y>0$ **then** $(y:= y-1.\ t:= t+1.\ P)$
else $(x:= x-1.\ (\exists n\cdot\ y:= n).\ t:= t+1.\ P)$

The execution time depends on x and on y and on the arbitrary values assigned to y . That means we need n to be nonlocal so we can refer to it in the specification P . But a nonlocal n would have a single arbitrary initial value that would be assigned to y every time x is decreased, whereas in our computation y may be assigned different arbitrary values every time x is decreased. So we change n into a function f of x . (Variable x never repeats a value; if it did repeat, we would have to make f be a function of time.)

Let f: $nat \rightarrow nat$. We say nothing more about f , so it is a completely arbitrary function from nat to nat . Introducing f gives us a way to refer to the arbitrary values, but does not say anything about when or how those arbitrary values are chosen. Let $s = \Sigma f[0;..x]$, which says s is the sum of the first x values of f . We prove

$t' = t+x+y+s \ \Leftarrow$ **if** $x=y=0$ **then** ok
else if $y>0$ **then** $(y:= y-1.\ t:= t+1.\ t' = t+x+y+s)$
else $(x:= x-1.\ y:= fx.\ t:= t+1.\ t' = t+x+y+s)$

The proof is in three cases.

$x=y=0 \land ok$
$\Rightarrow$ $x=y=s=0 \land t'=t$
$\Rightarrow$ $t' = t+x+y+s$

$y>0 \land (y{:=}\ y-1.\ \ t{:=}\ t+1.\ \ t' = t+x+y+s)$ substitution law twice
$=$ $y>0 \ \land\ t' = t+1+x+y-1+s$
$\Rightarrow$ $t' = t+x+y+s$

$x>0 \land y=0 \land (x{:=}\ x-1.\ \ y{:=}\ f\,x.\ \ t{:=}\ t+1.\ \ t' = t+x+y+s)$ substitution law 3 times
$=$ $x>0 \ \land\ y=0 \ \land\ t' = t+1+x-1+f\,(x-1)+\Sigma f\,[0;..x-1]$
$\Rightarrow$ $t' = t+x+y+s$

The execution time of the program is $x + y +$ (the sum of x arbitrary natural numbers) .

End of While Loop

5.2.1 Loop with Exit

Some languages provide a command to jump out of the middle of a loop. The syntax for a loop in such a language might be

loop P **end**

with the additional syntax

exit when b

allowed within P , where b is boolean. Sometimes the word "break" is used instead of "exit". As in Subsection 5.2.0, we consider refinement by a loop with exits to be an alternative notation. For example, if L is an implementable specification, then

L $\Leftarrow$ **loop**
  A.
  exit when b.
  C
end

is an alternative notation for

L $\Leftarrow$ A. **if** b **then** ok **else** $(C.\ L)$

Programmers who use loop constructs sometimes find that they reach their goal deep within several nested loops. The problem is how to get out. A boolean variable can be introduced for the purpose of recording whether the goal has been reached, and tested at each iteration of each level of loop to decide whether to continue or exit. Or a **go to** can be used to jump directly out of all the loops, saving all tests. Or perhaps the programming language provides a specialized **go to** for this purpose: **exit** n **when** b which means exit n loops when b is satisfied. For example, we may have something like this:

P $\Leftarrow$ **loop**
  A.
  loop
    B.
    exit 2 **when** c.
    D
  end.
  E
end

The refinement structure corresponding to this loop is

$P \Leftarrow A.\ Q$

$Q \Leftarrow B.$ **if** c **then** ok **else** $(D.\ Q)$

for some appropriately defined Q . It has often been suggested that every loop should have a specification, but the loop construct does not require it. The refinement structure does require it.

The preceding example had a deep exit but no shallow exit, leaving E stranded in a dead area. Here is an example with both deep and shallow exits.

```
P ⇐  loop
        A.
        exit 1 when b.
        C.
        loop
            D.
            exit 2 when e.
            F.
            exit 1 when g.
            H
        end.
        I
     end
```

The refinement structure corresponding to this loop is

$P \Leftarrow A.$ **if** b **then** ok **else** $(C.\ Q)$

$Q \Leftarrow D.$ **if** e **then** ok **else** $(F.$ **if** g **then** $(I.\ P)$ **else** $(H.\ Q))$

for some appropriately defined Q .

Loops with exits can always be translated easily to a refinement structure. But the reverse is not true; some refinement structures require the introduction of new variables and even whole data structures to encode them as loops with exits.

End of Exit Loop

5.2.2 Two-Dimensional Search

To illustrate the preceding subsection, we can do Exercise 157: Write a program to find a given item in a given 2-dimensional array. The execution time must be linear in the product of the dimensions.

Let the array be A , let its dimensions be n by m , and let the item we seek be x . We will indicate the position of x in A by the final values of natural variables i and j . If x occurs more than once, any of its positions will do. If it does not occur, we will indicate that by assigning i and j the values n and m respectively. The problem, except for time, is then P where

$P = $ **if** $x: A\ (0,..n)\ (0,..m)$ **then** $x = A\ i'\ j'$ **else** $i'=n \land j'=m$

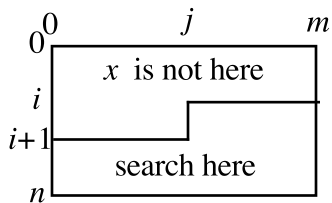
We may as well search row 0 first, then row 1 , and so on. Accordingly, we define Q to specify the search from row i onward:

$Q = $ **if** $x: A\ (i,..n)\ (0,..m)$ **then** $x = A\ i'\ j'$ **else** $i'=n \land j'=m$

Within each row, we search the columns in order, and so we define R to specify the search from row i column j onward:

$R = $ **if** $x: A\ i\ (j,..m),\ A\ (i+1,..n)\ (0,..m)$ **then** $x = A\ i'\ j'$ **else** $i'=n \land j'=m$

The expression $A\ i\ (j,..m),\ A\ (i+1,..n)\ (0,..m)$ represents the items in the bottom region of the following picture:



We now solve the problem in five easy pieces.

$P \Leftarrow i:= 0.\ i\le n \Rightarrow Q$

$i\le n \Rightarrow Q \Leftarrow$ **if** $i=n$ **then** $j:= m$ **else** $i<n \Rightarrow Q$

$i<n \Rightarrow Q \Leftarrow j:= 0.\ i<n \wedge j\le m \Rightarrow R$

$i<n \wedge j\le m \Rightarrow R \Leftarrow$ **if** $j=m$ **then** $(i:= i+1.\ i\le n \Rightarrow Q)$ **else** $i<n \wedge j<m \Rightarrow R$

$i<n \wedge j<m \Rightarrow R \Leftarrow$ **if** $A\ i\ j = x$ **then** ok **else** $(j:= j+1.\ i<n \wedge j\le m \Rightarrow R)$

It is easier to see the execution pattern when we retain only enough information for execution. The non-program specifications are needed for understanding the purpose, and for proof, but not for execution. To a compiler, the program appears as follows:

$P \Leftarrow i:= 0.\ L0$
$L0 \Leftarrow$ **if** $i=n$ **then** $j:= m$ **else** $(j:= 0.\ L1)$
$L1 \Leftarrow$ **if** $j=m$ **then** $(i:= i+1.\ L0)$
else if $A\ i\ j = x$ **then** ok
else $(j:= j+1.\ L1)$

In C, this is

```
      i = 0;
L0:  if (i==n) j = m;
      else {      j = 0;
            L1:  if (j==m) {i = i+1;  goto L0;}
                  else if (A[i][j]==x) ;
                  else {j = j+1;  goto L1;}
            }
```

To add recursive time, we put $t:= t+1$ just after $i:= i+1$ and after $j:= j+1$. Or, to be a little more clever, we can get away with a single time increment placed just before the test $j=m$. We also change the five specifications we are refining to refer to time. The time remaining is at most the area remaining to be searched.

$t' \le t + n\times m \Leftarrow i:= 0.\ i\le n \Rightarrow t' \le t + (n-i)\times m$

$i\le n \Rightarrow t' \le t + (n-i)\times m \Leftarrow$ **if** $i=n$ **then** $j:= m$ **else** $i<n \Rightarrow t' \le t + (n-i)\times m$

$i<n \Rightarrow t' \le t + (n-i)\times m \Leftarrow j:= 0.\ i<n \wedge j\le m \Rightarrow t' \le t + (n-i)\times m - j$

$i<n \wedge j\le m \Rightarrow t' \le t + (n-i)\times m - j \Leftarrow$
$t:= t+1.$
if $j=m$ **then** $(i:= i+1.\ i\le n \Rightarrow t' \le t + (n-i)\times m)$
else $i<n \wedge j<m \Rightarrow t' \le t + (n-i)\times m - j$

$i<n \wedge j<m \Rightarrow t' \le t + (n-i)\times m - j \Leftarrow$
if $A\ i\ j = x$ **then** ok
else $(j{:=}\ j+1.\ \ i<n \wedge j\le m \Rightarrow t' \le t + (n-i)\times m - j)$

End of Two-Dimensional Search

5.2.3 For Loop

Let us use the syntax

for $i{:=}\ m;..n$ **do** P

where i is a fresh name, m and n are integer expressions such that $m\le n$, and P is a specification, as an almost-typical notation for controlled iteration. The difference from popular languages is just that iteration continues up to but excluding $i=n$. To avoid some thorns, let us say also that i is not a state variable (so it cannot be assigned within P), and that the initial values of m and n control the iteration (so the number of iterations is $n-m$).

As with the previous loop constructs, we will not define the **for**-loop as a specification, but instead show how it is used in refinement. Let F be a function of two integer variables whose result is an implementable specification. Then

$Fmn \Leftarrow$ **for** $i{:=}\ m;..n$ **do** P

is an abbreviation of the three refinements

$Fii \Leftarrow m\le i\le n \wedge ok$
$Fi(i+1) \Leftarrow m\le i<n \wedge P$
$Fik \Leftarrow m\le i<j<k\le n \wedge (Fij.\ Fjk)$

If $m=n$ there are no iterations, and specification Fmn must be satisfied by doing nothing ok . The body of the loop has to do one iteration $Fi(i+1)$. Finally, Fmn must be satisfied by first doing the iterations from m to an intermediate index j , and then doing the rest of the iterations from j to n .

For example, let the state consist of integer variable x , and let F be defined as

$F = \langle i, j{:}\ nat \rightarrow x' = x\times 2^{j-i}\rangle$

Then we can solve the exponentiation problem $x'=2^n$ in two refinements:

$x'=2^n \Leftarrow x{:=}\ 1.\ \ F0n$
$F0n \Leftarrow$ **for** $i{:=}\ 0;..n$ **do** $x{:=}\ 2\times x$

The first refinement is proven by the Substitution Law. To prove the second, we must prove three theorems

$Fii \Leftarrow 0\le i\le n \wedge ok$
$Fi(i+1) \Leftarrow 0\le i<n \wedge (x{:=}\ 2\times x)$
$Fik \Leftarrow 0\le i<j<k\le n \wedge (Fij.\ Fjk)$

all of which are easy.

The recursive time measure requires each loop to contain a time increment of at least one time unit. In general, the time taken by the body of a **for** loop may be a function f of the iteration i . Using $t' = t + \Sigma i{:}\ m,..n\cdot fi$ as **for**-loop specification Fmn , the **for**-loop rule tells us

$t' = t + \Sigma i{:}\ m,..n\cdot fi \Leftarrow$ **for** $i{:=}\ m;..n$ **do** $t' = t+fi$

When the body takes constant time c , this simplifies to

$t' = t + (n-m)\times c \Leftarrow$ **for** $i{:=}\ m;..n$ **do** $t' = t+c$

A typical use of the **for**-loop rule is to do something to each item in a list. For example, Exercise 268 asks us to add 1 to each item in a list. The specification is

$$\#L'=\#L \ \land \ \forall i\colon 0,..\#L\cdot L'i=Li+1$$

Now we need a specification Fik that describes an arbitrary segment of iterations: adding 1 to each item from index i to index k .

$$Fik \ = \ \#L'=\#L \land (\forall j\colon i,..k\cdot L'j=Lj+1) \land (\forall j\colon (0,..i), (k,..\#L)\cdot L'j=Lj)$$

To prove

$$F\,0\,(\#L) \ \Leftarrow \ \textbf{for } i{:=}\ 0;..\#L \textbf{ do } L{:=}\ i{\rightarrow}Li+1 \mid L$$

we must prove three theorems:

$$Fii \ \Leftarrow \ 0{\le}i{\le}\#L \land ok$$
$$Fi(i+1) \ \Leftarrow \ 0{\le}i{<}\#L \land (L{:=}\ i{\rightarrow}Li+1 \mid L)$$
$$Fik \ \Leftarrow \ 0{\le}i{<}j{<}k{\le}\#L \land (Fij.\ Fjk)$$

Sometimes the **for**-loop specification Fmn has the form $Im{\Rightarrow}I'n$, where I is a function of one variable whose result is a precondition, and I' is the function whose result is the corresponding postcondition. When I is applied to the **for**-loop index, condition Ii is called an invariant. An advantage of this form of specification is that both $Fii \Leftarrow ok$ and $Fik \Leftarrow (Fij.\ Fjk)$ are automatically satisfied. Not all **for**-loop specifications can be put in this form; neither the timing nor the previous example (add 1 to each item) can be. But the earlier exponential example can be put in this form. Define

$$I \ = \ \langle i\colon nat{\rightarrow}x=2^i\rangle$$

Then the solution is

$$x'=2^n \ \Leftarrow \ x{:=}\ 1.\ I0{\Rightarrow}I'n$$
$$I0{\Rightarrow}I'n \ \Leftarrow \ \textbf{for } i{:=}\ 0;..n \textbf{ do } Ii{\Rightarrow}I'(i+1)$$
$$Ii{\Rightarrow}I'(i+1) \ \Leftarrow \ x{:=}\ 2{\times}x$$

As another example of the invariant form of the **for**-loop rule, here is Exercise 186(a): Given a list of integers, possibly including negatives, write a program to find the minimum sum of any segment (sublist of consecutive items). Let L be the list. Formally, the problem is P where

$$P \ = \ s' \ = \ MIN\ i, j\cdot \Sigma\, L\,[i;..j]$$

where $0 \le i \le j \le \#L$. The condition $I\,k$ will say that s is the minimum sum of any segment up to index k . For $k{=}0$ there is only one segment, the empty segment, and its sum is 0 . When $k{=}\#L$ all segments are included and we have the desired result. To go from $I\,k$ to $I\,(k+1)$ we have to consider those segments that end at index $k+1$. We could find the sum of each new segment, then take the minimum of those sums and of s to be the new value of s . But we can do better. Each segment ending at index $k+1$ is a one-item extension of a segment ending at index k with one exception: the empty segment ending at $k+1$.

$$[\ 4\ ;\ -2\ ;\ -8\ \overset{k}{\downarrow};\ 7\ \overset{k+1}{\downarrow};\ 3\ ;\ 0\ ;\ -1\ ]$$

If we know the minimum sum c of any segment ending at k , then $min\ (c + L\ k)\ 0$ is the minimum sum of any segment ending at $k+1$. So we define, for $0 \le k \le \#L$,

$$I\,k \ = \ \ s \ = \ (MIN\ i\colon 0,..k+1\cdot MIN\ j\colon i,..k+1\cdot \Sigma\, L\,[i;..j])$$
$$\land\ \ c \ = \ (MIN\ i\colon 0,..k+1\cdot \Sigma\, L\,[i;..k])$$

Now the program is easy.

$$P \ \Leftarrow \ s{:=}\ 0.\ c{:=}\ 0.\ I\,0 \Rightarrow I'(\#L)$$
$$I\,0 \Rightarrow I'(\#L) \ \Leftarrow \ \textbf{for } k{:=}\ 0;..\#L \textbf{ do } I\,k \Rightarrow I'(k+1)$$
$$I\,k \Rightarrow I'(k+1) \ \Leftarrow \ c{:=}\ min\ (c + L\,k)\,0.\ s{:=}\ min\ c\ s$$

End of For Loop

5.2.4 Go To

Programming texts often warn that the **go to** is harmful, and should be avoided, but it causes no more problem for proof than loop constructs. For example, suppose the fast exponentiation program $z'=x^y$ of Subsection 4.2.6 were written as follows (using colon for labeling).

A: $z:= 1$. **if** *even* y **then go to** C
  else B: ($z:= z\times x$. $y:= y-1$.
    C: **if** $y=0$ **then go to** E
     else D: ($x:= x\times x$. $y:= y/2$. **if** *even* y **then go to** D **else go to** B)).

E:

Straight from the program, what needs to be proven is the following:

$A \Leftarrow$ $z:= 1$. **if** *even* y **then** C **else** B
$B \Leftarrow$ $z:= z\times x$. $y:= y-1$. C
$C \Leftarrow$ **if** $y=0$ **then** E **else** D
$D \Leftarrow$ $x:= x\times x$. $y:= y/2$. **if** *even* y **then** D **else** B

for appropriate formalizations of the labels (specifically, $A = z'=x^y$, $B = odd\ y \Rightarrow z'=z\times x^y$, $C = even\ y \Rightarrow z'=z\times x^y$, $D = even\ y \wedge y>0 \Rightarrow z'=z\times x^y$, and $E = ok$). The difficulty with **go to** , as with loop constructs, is inventing the specifications.

End of Go To

End of Control Structures

5.3 Time and Space Dependence

Some programming languages provide a clock, or a delay, or other time-dependent features. Our examples have used the time variable as a ghost, or auxiliary variable, never affecting the course of a computation. It was used as part of the theory, to prove something about the execution time. Used for that purpose only, it did not need representation in a computer. But if there is a readable clock available as a time source during a computation, it can be used to affect the computation. The assignment $deadline:= t + 5$ is allowed, as is **if** $t \le deadline$ **then** ... **else** ... . But the assignment $t:= 5$ is not allowed. We can look at the clock, but not reset it arbitrarily; all clock changes must correspond to the passage of time (according to some measure). (A computer operator may need to set the clock sometimes, but that is not part of the theory of programming.)

We may occasionally want to specify the passage of time. For example, we may want the computation to “wait until time w ”. Let us invent a notation for it, and define it formally as

wait until w $=$ $t:= max\ t\ w$

Because we are not allowed to reset the clock, $t:= max\ t\ w$ is not acceptable as a program until we refine it. Letting time be an extended integer and using recursive time,

wait until w $\Leftarrow$ **if** $t\ge w$ **then** *ok* **else** ($t:= t+1$. **wait until** w)

and we obtain a busy-wait loop. We can prove this refinement by cases. First,

$t\ge w \wedge ok$
$=$ $t\ge w \wedge (t:= t)$
$\Rightarrow$ $t:= max\ t\ w$

And second,

$t<w \wedge (t:= t+1.\ t:= max\ t\ w)$
  In the left conjunct, use $t: xint$. In the right conjunct, use the Substitution Law.
$=$ $t+1 \le w \wedge (t:= max\ (t+1)\ w)$
$=$ $t+1 \le w \wedge (t:= w)$
$=$ $t<w \wedge (t:= max\ t\ w)$
$\Rightarrow$ $t:= max\ t\ w$

In programs that depend upon time, we should use the real time measure, rather than the recursive time measure. We also need to be more careful where we place our time increments. And we need a slightly different definition of **wait until** w , but we leave that as Exercise 275(b).

Our space variable s , like the time variable t , has so far been used to prove things about space usage, not to affect the computation. But if a program has space usage information available to it, there is no harm in using that information. Like t , s can be read but not written arbitrarily. All changes to s must correspond to changes in space usage.

End of Time and Space Dependence

5.4 Assertions optional

5.4.0 Checking

As a safety check, some programming languages include the notation

assert b

where b is boolean, to mean something like "I believe b is true". If it comes at the beginning of a procedure or method, it may use the word **precondition** ; if at the end, it may use the word **postcondition** ; if it comes at the start or end of a loop, it may use the word **invariant** ; these are all the same construct. It is executed by checking that b is true; if it is, execution continues normally, but if not, an error message is printed and execution is suspended. The intention is that in a correct program, the asserted expressions will always be true, and so all assertions are redundant. All error checking requires redundancy, and assertions help us to find errors and prevent subsequent damage to the state variables. But it's not free; it costs execution time.

Assertions are defined as follows.

assert b = **if** b **then** ok **else** ($print$ "error". **wait until** ∞)

If b is true, **assert** b is the same as ok . If b is false, execution cannot proceed in finite time to any following actions. Assertions are an easy way to make programs more robust.

End of Checking

5.4.1 Backtracking

If P and Q are implementable specifications, so is $P \lor Q$. The disjunction can be implemented by choosing one of P or Q and satisfying it. Normally this choice is made as a refinement, either $P \lor Q \Leftarrow P$ or $P \lor Q \Leftarrow Q$. We could save this programming step by making disjunction a programming connective, perhaps using the notation **or** . For example,

$x := 0$ **or** $x := 1$

would be a program whose execution assigns either 0 or 1 to x . This would leave the choice of disjunct to the programming language implementer.

The next construct radically changes the way we program. We introduce the notation

ensure b

where b is boolean, to mean something like "make b be true". We define it as follows.

ensure b = **if** b **then** ok **else** $b' \land ok$

= $b' \land ok$

Like **assert** b , **ensure** b is equal to ok if b is true. But when b is false, there is a problem: the computation must make b true without changing anything. This is unimplementable (unless b is identically true). However, in combination with other constructs, the whole may be implementable. Consider the following example in variables x and y .

$x:= 0$ **or** $x:= 1$. **ensure** $x=1$
$=$ $\exists x'', y''\cdot (x''=0 \land y''=y \lor x''=1 \land y''=y) \land x'=1 \land x'=x'' \land y'=y''$
$=$ $x'=1 \land y'=y$
$=$ $x:= 1$

Although an implementation is given a choice between $x:= 0$ and $x:= 1$, it must choose the right one to satisfy a later condition. It can do so by making either choice (as usual), and when faced with a later **ensure** whose condition is false, it must backtrack and make another choice. Since choices can be nested within choices, a lot of bookkeeping is necessary.

Several popular programming languages, such as Prolog, feature backtracking. They may state that choices are made in a particular order (we have omitted that complication). Two warnings should accompany such languages. First, it is the programmer's responsibility to show that a program is implementable; the language does not guarantee it. Alternatively, the implementation does not guarantee that computations will satisfy the program, since it is sometimes impossible to satisfy it. The second warning is that the time and space calculations do not work.

End of Backtracking

End of Assertions

5.5 Subprograms

5.5.0 Result Expression

Let P be a specification and e be an expression in unprimed variables. Then

P **result** e

is an expression of the initial state. It expresses the result of executing P and then evaluating e . For example, the following expresses an approximation to the base of the natural logarithms.

var *term*, *sum*: *rat* := 1·
for *i*:= 1;..15 **do** (*term*:= *term*/*i*. *sum*:= *sum*+*term*)
result *sum*

The axiom for the **result** expression is

$x' = (P$ **result** $e) \;=\; P.\ x'=e$

where x is any state variable of the right type.

The example introduces local variables *term* and *sum* , and no other variables are reassigned. So clearly the nonlocal state is unchanged. But consider

$y:= y+1$ **result** y

The result is as if the assignment $y:= y+1$ were executed, then y is the result, except that the value of variable y is unchanged.

$x:= (y:= y+1$ **result** $y)$
$=$ $x' = (y:= y+1$ **result** $y) \land y'=y$
$=$ $(y:= y+1.\ x'=y) \land y'=y$
$=$ $x' = y+1 \land y'=y$
$=$ $x:= y+1$

The expression P **result** e can be implemented as follows. Replace each nonlocal variable within P and e that is assigned within P by a fresh local variable initialized to the value of the nonlocal variable. Then execute P and evaluate e . In the implementation of some programming languages, the introduction of fresh local variables for this purpose is not done, so the evaluation of an expression may cause a state change. State changes resulting from the evaluation of an expression are called "side-effects". With side-effects, mathematical reasoning is not possible. For example,

we cannot say $x+x = 2\times x$, nor even $x=x$, since x might be $(y:= y+1$ **result** $y)$, and each evaluation results in an integer that is 1 larger than the previous evaluation. Side effects are easily avoided; a programmer can introduce the necessary local variables if the language implementation fails to do so. Some programming languages forbid assignments to nonlocal variables within expressions, so the programmer is required to introduce the necessary local variables.

If a programming language allows side-effects, we have to get rid of them before using any theory. For example,

$x:= (P$ **result** $e)$ becomes $(P.\ x:= e)$

after renaming local variables within P as necessary to avoid clashes with nonlocal variables, and allowing the scope of variables declared in P to extend through $x:= e$. For another example,

$x:= y + (P$ **result** $e)$ becomes (**var** $z:= y\cdot\ P.\ x:= z+e)$

with similar provisos.

The recursive time measure that we have been using neglects the time for expression evaluation. This is reasonable in some applications for expressions consisting of a few operations implemented in computer hardware. For expressions using operations not implemented in hardware (perhaps list catenation) it is questionable. For **result** expressions containing loops, it is unreasonable. But allowing a **result** expression to increase a time variable would be a side-effect, so here is what we do. We first include time in the **result** expression for the purpose of calculating a time bound. Then we remove the time variable from the result expression (to get rid of the side-effect) and we put a time increment in the program that uses the **result** expression.

End of Result Expression

5.5.1 Function

In many popular programming languages, a function is a combination of assertion about the result, name of the function, parameters, scope control, and **result** expression. It's a "package deal". For example, in C, the binary exponential function looks like this:

```
int bexp (int n)
{ int r = 1;
  int i;
  for (i=0; i<n; i++) r = r*2;
  return r; }
```

In our notations, this would be

$bexp$ = ⟨ n: int→
  var r: int := 1·
  for i:= 0;..n **do** r:= r×2.
  assert r: int
  result r ⟩

We present these programming features separately so that they can be understood separately. They can be combined in any way desired, as in the example. The harm in providing one construct for the combination is its complexity. Programmers trained with these languages may be unable to separate the issues and realize that naming, parameterization, assertions, local scope, and **result** expressions are independently useful.

Even the form of function we are using in this book could be both simplified and generalized. Stating the domain of a parameter is a special case of axiom introduction, which can be separated from name introduction (see Exercise 90).

End of Function

5.5.2 Procedure

The procedure (or void function, or method), as it is found in many languages, is a “package deal” like the function. It combines name declaration, parameterization, and local scope. The comments of the previous subsection apply here too. There are also some new issues.

To use our theory for program development, not just verification, we must be able to talk about a procedure whose body is an unrefined specification, not yet a program. For example, we may want a procedure P with parameter x defined as

$$P \;=\; \langle x\colon int \rightarrow a' < x < b' \rangle$$

that assigns variables a and b values that lie on opposite sides of a value to be supplied as argument. We can use procedure P before we refine its body. For example,

$$P\,3 \;=\; a' < 3 < b'$$
$$P\,(a+1) \;=\; a' < a+1 < b'$$

The body is easily refined as

$$a' < x < b' \;\Leftarrow\; a := x-1.\; b := x+1$$

Our choice of refinement does not alter our definition of P ; it is of no use when using P . The users don't need to know the implementation, and the implementer doesn't need to know the uses.

A procedure and argument can be translated to a local variable and initial value.

$$\langle p\colon D \rightarrow B \rangle\, a \;=\; (\mathbf{var}\; p\colon D := a \cdot\; B) \qquad \text{if } B \text{ doesn't use } p' \text{ or } p{:=}$$

This translation suggests that a parameter is really just a local variable whose initial value will be supplied as an argument. In many popular programming languages, that is exactly the case. This is an unfortunate confusion of specification and implementation. The decision to create a parameter, and the choice of its domain, are part of a procedural specification, and are of interest to a user of the procedure. The decision to create a local variable, and the choice of its domain, are normally part of refinement, part of the process of implementation, and should not be of concern to a user of the procedure. When a parameter is assigned a value within a procedure body, it is acting as a local variable and no longer has any connection to its former role as parameter.

Another kind of parameter, usually called a reference parameter or **var** parameter, stands for a nonlocal variable to be supplied as argument. Here is an example, using $\langle *\;\rangle$ to introduce a reference parameter.

$$\langle *x\colon int \rightarrow a := 3.\; b := 4.\; x := 5 \rangle\, a$$
$$=\quad a := 3.\; b := 4.\; a := 5$$
$$=\quad a'=5 \wedge b'=4$$

Reference parameters can be used only when the body of the procedure is pure program, not using any other specification notations. For the above example, if we had written

$$\langle *x\colon int \rightarrow a'=3 \wedge b'=4 \wedge x'=5 \rangle\, a$$

we could not just replace x with a , nor even x' with a' . Furthermore, we cannot do any reasoning about the procedure body until after the procedure has been applied to its arguments. The following example has a procedure body that is equivalent to the previous example,

$$\langle *x\colon int \rightarrow x := 5.\; b := 4.\; a := 3 \rangle\, a$$
$$=\quad a := 5.\; b := 4.\; a := 3$$
$$=\quad a'=3 \wedge b'=4$$

but the result is different. Reference parameters prevent the use of specification, and they prevent any reasoning about the procedure by itself. We must apply our programming theory separately for each call. This contradicts the purpose of procedures.

End of Procedure

End of Subprograms

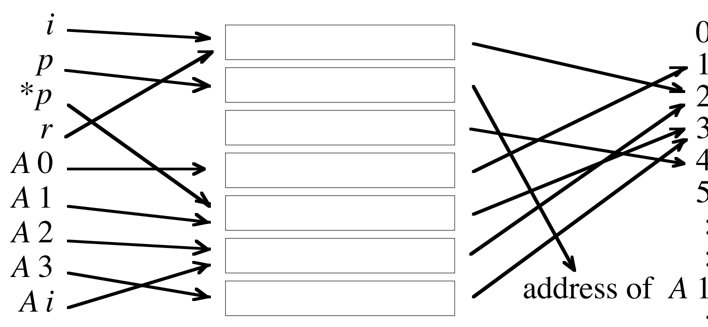
5.6 Alias optional

Many popular programming languages present us with a model of computation in which there is a memory consisting of a large number of individual storage cells. Each cell contains a value. Via the programming language, cells have names. Here is a standard sort of picture.

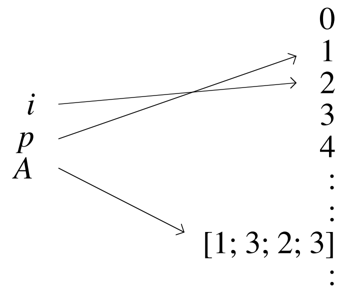
names	cell
r, i	2
p	address of $A\ 1$
	4
$A\ 0$	1
$*p$, $A\ 1$	3
$A\ i$, $A\ 2$	2
$A\ 3$	3

In the picture, p is a pointer variable that currently points to array element $A\ 1$, and $*p$ is p dereferenced; so $*p$ and $A\ 1$ refer to the same memory cell. Since variable i currently has value 2 , $A\ i$ and $A\ 2$ refer to the same cell. And r is a reference parameter for which variable i has been supplied as argument, so r and i refer to the same cell. We see that a cell may have zero, one, two, or more names. When a cell has two or more names that are visible at the same time, the names are said to be "aliases".

As we have seen with arrays and with reference parameters, aliasing prevents us from applying our theory of programming. Some programming languages prohibit aliasing. Unfortunately, aliasing is difficult to detect, especially during program construction before a specification has been fully refined as a program. To most people, prohibitions and restrictions are distasteful. To avoid the prohibition, we have a choice: we can complicate our theory of programming to handle aliasing, or we can simplify our model of computation to eliminate it. If we redraw our picture slightly, we see that there are two mappings: one from names to cells, and one from cells to values.



An assignment such as $p:=$ address of $A\ 3$ or $i:= 4$ can change both mappings at once. An assignment to one name can change the value indirectly referred to by another name. To simplify the picture and eliminate the possibility of aliasing, we eliminate the cells and allow a richer space of values. Here is the new picture.



Pointer variables can be replaced by index variables dedicated to one structure so that they can be implemented as addresses. Reference parameters are unnecessary if functions can return structured values. The simpler picture is perfectly adequate, and the problem of aliasing disappears.

End of Alias

5.7 Probabilistic Programming optional

Probability Theory has been developed using the arbitrary convention that a probability is a real number between 0 and 1 inclusive

$prob = \S r: real\cdot 0\le r\le 1$

with 1 representing "certainly true", 0 representing "certainly false", 1/2 representing "equally likely true or false", and so on. Accordingly, for this section only, we add the axioms

$\top = 1$

$\bot = 0$

With these axioms, boolean operators can be expressed arithmetically. For example, $\neg x = 1-x$, $x\wedge y = x\times y$, and $x\vee y = x - x\times y + y$.

A distribution is an expression whose value (for all assignments of values to its variables) is a probability, and whose sum (over all assignments of values to its variables) is 1 . (For the sake of simplicity, we consider only distributions over boolean and integer variables; for rational and real variables, summations become integrals.) For example, if $n: nat+1$, then 2^{-n} is a distribution because

$(\forall n: nat+1\cdot 2^{-n}: prob) \wedge (\Sigma n: nat+1\cdot 2^{-n})=1$

If we have two variables $n, m: nat+1$, then 2^{-n-m} is a distribution because

$(\forall n, m: nat+1\cdot 2^{-n-m}: prob) \wedge (\Sigma n, m: nat+1\cdot 2^{-n-m})=1$

A distribution can be used to tell the frequency of occurrence of values of its variables. For example, 2^{-n} says that n has value 3 one-eighth of the time. Distribution 2^{-n-m} says that the state in which n has value 3 and m has value 1 occurs one-sixteenth of the time. A distribution can also be used to say what values we expect or predict variables to have. Distribution 2^{-n} says that n is equally as likely to have the value 1 as it is to not have the value 1 , and twice as likely to have the value 1 as it is to have the value 2 . A distribution can also be used to specify the probability that we want for the value of each variable.

Suppose we have one natural state variable n . The specification $n' = n+1$ tells us that, for any given initial value n , the final value n' is $n+1$. Stated differently, it says the final value n' equals the initial value $n+1$ with probability 1 , and equals any other value with probability 0 . For any values of n and n' , the value of $n' = n+1$ is either $\top$ (1) or $\bot$ (0), so it is a probability. The specification $n' = n+1$ is not a distribution of n and n' because there are infinitely many pairs of values that give $n' = n+1$ the value $\top$ or 1 , and so

$\Sigma n, n'\cdot n' = n+1 = \infty$

But for any fixed value of n , there is a single value of n' that gives $n' = n+1$ the value $\top$ or 1 ,

and so

$$\Sigma n'\cdot\ n' = n+1 \quad = \quad 1$$

For any fixed value of n , $n' = n+1$ is a one-point distribution of n' . Similarly, any implementable deterministic specification is a one-point distribution of the final state.

We generalize our programming notations to allow probabilistic operands as follows.

$$\begin{array}{lll} ok & = & (x'=x) \times (y'=y) \times ... \\ x{:=}\ e & = & (x'=e) \times (y'=y) \times ... \\ \textbf{if } b \textbf{ then } P \textbf{ else } Q & = & b \times P \ + \ (1-b) \times Q \\ P.\,Q & = & \Sigma x'', y'', ...\cdot \quad (\text{for } x', y', ... \text{ substitute } x'', y'', ... \text{ in } P) \\ & & \qquad \times \ (\text{for } x, y, ... \text{ substitute } x'', y'', ... \text{ in } Q) \end{array}$$

Since $\bot=0$ and $\top=1$, the definitions of ok and assignment have not changed; they have just been expressed arithmetically. If b , P , and Q are boolean, the definitions of **if** b **then** P **else** Q and $P.Q$ have not changed. But now they apply not only to $\bot$ and $\top$, that is to 0 and 1 , but also to values between 0 and 1 . In other words, they apply to probabilities. If b is a probability of the initial state, and P and Q are distributions of the final state, then **if** b **then** P **else** Q is a distribution of the final state. If P and Q are distributions of the final state, then $P.Q$ is a distribution of the final state. For example,

if 1/3 **then** $x{:=}\ 0$ **else** $x{:=}\ 1$

means that with probability $1/3$ we assign x the value 0 , and with the remaining probability $2/3$ we assign x the value 1 . In one variable x ,

$$\begin{array}{ll} & \textbf{if } 1/3 \textbf{ then } x{:=}\ 0 \textbf{ else } x{:=}\ 1 \\ = & 1/3 \times (x'=0) + (1 - 1/3) \times (x'=1) \end{array}$$

Let us evaluate this expression using the value 0 for x' .

$$\begin{array}{ll} & 1/3 \times (0=0) + (1 - 1/3) \times (0=1) \\ = & 1/3 \times 1 + 2/3 \times 0 \\ = & 1/3 \end{array}$$

which is the probability that x has final value 0 . Let us evaluate this expression using the value 1 for x' .

$$\begin{array}{ll} & 1/3 \times (1=0) + (1 - 1/3) \times (1=1) \\ = & 1/3 \times 0 + 2/3 \times 1 \\ = & 2/3 \end{array}$$

which is the probability that x has final value 1 . Let us evaluate this expression using the value 2 for x' .

$$\begin{array}{ll} & 1/3 \times (2=0) + (1 - 1/3) \times (2=1) \\ = & 1/3 \times 0 + 2/3 \times 0 \\ = & 0 \end{array}$$

which is the probability that x has final value 2 .

Here is a slightly more elaborate example in one variable x .

$$\begin{array}{ll} & \textbf{if } 1/3 \textbf{ then } x{:=}\ 0 \textbf{ else } x{:=}\ 1. \\ & \textbf{if } x{=}0 \textbf{ then if } 1/2 \textbf{ then } x{:=}\ x+2 \textbf{ else } x{:=}\ x+3 \\ & \textbf{else if } 1/4 \textbf{ then } x{:=}\ x+4 \textbf{ else } x{:=}\ x+5 \\ = & \Sigma x''\cdot \quad ((x''=0)/3 + (x''=1)\times 2/3) \\ & \qquad \times \ (\ (x''=0) \times ((x' = x''+2)/2 + (x' = x''+3)/2) \\ & \qquad\quad + \ (1 - (x''=0)) \times ((x' = x''+4)/4 + (x' = x''+5)\times 3/4)) \\ = & (x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2 \end{array}$$

After the first line, x might be 0 or 1 . If it is 0 , then with probability $1/2$ we add 2 , and with the remaining probability $1/2$ we add 3 ; otherwise (if x is not 0) with probability $1/4$ we add 4 and with the remaining probability $3/4$ we add 5 . The sum is much easier than it looks

because all values for x'' other than 0 and 1 make a 0 contribution to the sum. The final line says that the resulting value of variable x is 2 with probability $1/6$, 3 with probability $1/6$, 5 with probability $1/6$, 6 with probability $1/2$, and any other value with probability 0 .

Let P be any distribution of final states, and let e be any number expression over initial states. After execution of P , the average value of e is $(P.e)$. For example, the average value of n^2 as n varies over $nat+1$ according to distribution 2^{-n} is

$$
\begin{array}{ll}
 & 2^{-n'}.\ n^2 \\
= & \Sigma n'': nat+1\cdot\ 2^{-n''} \times n''^2 \\
= & 6
\end{array}
$$

After execution of the previous example, the average value of x is

if $1/3$ **then** $x:= 0$ **else** $x:= 1$.
if $x=0$ **then if** $1/2$ **then** $x:= x+2$ **else** $x:= x+3$
else if $1/4$ **then** $x:= x+4$ **else** $x:= x+5$.

$$
\begin{array}{ll}
 & x \\
= & (x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2.\ x \\
= & \Sigma x''\cdot\ ((x''=2)/6 + (x''=3)/6 + (x''=5)/6 + (x''=6)/2) \times x'' \\
= & 1/6 \times 2 + 1/6 \times 3 + 1/6 \times 5 + 1/2 \times 6 \\
= & 4 + 2/3
\end{array}
$$

Let P be any distribution of final states, and let b be any boolean expression over initial states. After execution of P , the probability that b is true is $(P.b)$. Probability is just the average value of a boolean expression. For example, after execution of the previous example, the probability that x is greater than 3 is

if $1/3$ **then** $x:= 0$ **else** $x:= 1$.
if $x=0$ **then if** $1/2$ **then** $x:= x+2$ **else** $x:= x+3$
else if $1/4$ **then** $x:= x+4$ **else** $x:= x+5$.

$$
\begin{array}{ll}
 & x>3 \\
= & (x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2.\ x>3 \\
= & \Sigma x''\cdot\ ((x''=2)/6 + (x''=3)/6 + (x''=5)/6 + (x''=6)/2) \times (x''>3) \\
= & 1/6 \times (2>3) + 1/6 \times (3>3) + 1/6 \times (5>3) + 1/2 \times (6>3) \\
= & 2/3
\end{array}
$$

Most of the laws, including all distribution laws and the Substitution Law, apply without change to probabilistic specifications and programs. For example, the previous two calculations could begin by distributing the final line (x in the first one, $x>3$ in the second) back into the **then**- and **else**-parts that increase x , then distribute **if** $x=0$ ... back into the **then**- and **else**-parts that initialize x , then use the Substitution Law six times, thus avoiding the need to sum.

5.7.0 Random Number Generators

Many programming languages provide a random number generator (sometimes called a “pseudo-random number generator”). The usual notation is functional, and the usual result is a value whose distribution is uniform (constant) over a nonempty finite range. If $n: nat+1$, we use the notation $rand\ n$ for a generator that produces natural numbers uniformly distributed over the range $0,..n$. So $rand\ n$ has value r with probability $(r: 0,..n) / n$.

Functional notation for a random number generator is inconsistent. Since $x=x$ is a law, we should be able to simplify $rand\ n = rand\ n$ to $\top$, but we cannot because the two occurrences of $rand\ n$ might generate different numbers. Since $x+x = 2\times x$ is a law, we should be able to simplify $rand\ n + rand\ n$ to $2 \times rand\ n$, but we cannot. To restore consistency, we replace each use of $rand$ with a fresh variable before we do anything else. We can replace $rand\ n$ with integer variable r whose value has probability $(r\text{: } 0,..n) / n$. Or, if you prefer, we can replace $rand\ n$ with variable $r\text{: } 0,..n$ whose value has probability $1/n$. (This is a mathematical variable, or in other words, a state constant; there is no r'.) For example, in one state variable x,

$$
\begin{array}{lll}
 & x{:=}\ rand\ 2.\ \ x{:=}\ x + rand\ 3 & \text{replace one } rand \text{ with } r \text{ and one with } s \\
= & \Sigma r\text{: } 0,..2\cdot \Sigma s\text{: } 0,..3\cdot (x{:=}\ r)/2.\ \ (x{:=}\ x + s)/3 & \text{Substitution Law} \\
= & \Sigma r\text{: } 0,..2\cdot \Sigma s\text{: } 0,..3\cdot (x' = r{+}s) / 6 & \text{sum} \\
= & ((x' = 0{+}0) + (x' = 0{+}1) + (x' = 0{+}2) + (x' = 1{+}0) + (x' = 1{+}1) + (x' = 1{+}2)) / 6 & \\
= & (x'{=}0) / 6\ +\ (x'{=}1) / 3 + (x'{=}2) / 3\ +\ (x'{=}3) / 6 &
\end{array}
$$

which says that x' is 0 with probability $1/6$, 1 with probability $1/3$, 2 with probability $1/3$, 3 with probability $1/6$, and any other value with probability 0.

Whenever $rand$ occurs in the context of a simple equation, such as $r = rand\ n$, we don't need to introduce a variable for it, since one is supplied. We just replace the deceptive equation with $(r\text{: } 0,..n) / n$. For example, in one variable x,

$$
\begin{array}{lll}
 & x{:=}\ rand\ 2.\ \ x{:=}\ x + rand\ 3 & \text{replace assignments} \\
= & (x'\text{: } 0,..2)/2.\ \ (x'\text{: } x{+}(0,..3))/3 & \text{dependent composition} \\
= & \Sigma x''\cdot (x''\text{: } 0,..2)/2 \times (x'\text{: } x''{+}(0,..3))/3 & \text{sum} \\
= & 1/2 \times (x'\text{: } 0,..3)/3\ +\ 1/2 \times (x'\text{: } 1,..4)/3 & \\
= & (x'{=}0) / 6\ +\ (x'{=}1) / 3 + (x'{=}2) / 3\ +\ (x'{=}3) / 6 &
\end{array}
$$

as before. And **if** $rand\ 2$ **then** A **else** B can be replaced by **if** $1/2$ **then** A **else** B.

Although $rand$ produces uniformly distributed natural numbers, it can be transformed into many different distributions. We just saw that $rand\ 2 + rand\ 3$ has value n with distribution $((n{=}0) + (n{=}3)) / 6\ +\ ((n{=}1) + (n{=}2)) / 3$. As another example, $rand\ 8 < 3$ has boolean value b with distribution

$$
\begin{array}{ll}
 & \Sigma r\text{: } 0,..8\cdot (b = (r{<}3)) / 8 \\
= & (b{=}\top) \times 3/8\ +\ (b{=}\bot) \times 5/8 \\
= & 5/8 - b/4
\end{array}
$$

which says that b is $\top$ with probability $3/8$, and $\bot$ with probability $5/8$.

Exercise 281 is a simplified version of blackjack. You are dealt a card from a deck; its value is in the range 1 through 13 inclusive. You may stop with just one card, or have a second card if you want. Your object is to get a total as near as possible to 14, but not over 14. Your strategy is to take a second card if the first is under 7. Assuming each card value has equal probability (actually, the second card drawn has a diminished probability of having the same value as the first card drawn, but let's ignore that complication), we represent a card as $(rand\ 13) + 1$. In one variable x, the game is

$$
\begin{array}{lll}
 & x{:=}\ (rand\ 13) + 1.\ \ \textbf{if}\ x{<}7\ \textbf{then}\ x{:=}\ x + (rand\ 13) + 1\ \textbf{else}\ ok & \\
 & & \text{replace } rand \text{ and } ok \\
= & (x'\text{: } (0,..13){+}1)/13.\ \ \textbf{if}\ x{<}7\ \textbf{then}\ (x'\text{: } x{+}(0,..13){+}1)/13\ \textbf{else}\ x'{=}x & \text{replace . and } \textbf{if} \\
= & \Sigma x''\cdot (x''\text{: } 1,..14)/13 \times ((x''{<}7){\times}(x'\text{: } x''{+}1,..x''{+}14)/13 + (x''{\ge}7){\times}(x'{=}x'')) & \\
 & & \text{by several omitted steps} \\
= & ((2{\le}x'{<}7){\times}(x'{-}1) + (7{\le}x'{<}14){\times}19 + (14{\le}x'{<}20){\times}(20{-}x')) / 169 &
\end{array}
$$

That is the distribution of x' if we use the "under 7" strategy. We can similarly find the distribution of x' if we use the "under 8" strategy, or any other strategy. But which strategy is best? To compare two strategies, we play both of them at once. Player x will play "under n" and player y will play "under $n+1$" using exactly the same cards c and d (the result would be no different if they used different cards, but it would require more variables). Here is the new game, followed by the condition that x wins:

$$c:= (rand\ 13) + 1.\ \ d:= (rand\ 13) + 1.$$
$$\textbf{if } c < n \textbf{ then } x:= c+d \textbf{ else } x:= c.\ \ \textbf{if } c < n+1 \textbf{ then } y:= c+d \textbf{ else } y:= c.$$
$$y<x\le 14 \lor x\le 14<y \qquad \text{Replace } rand \text{ and use the Functional-Imperative Law twice.}$$
$$= \quad (c': (0,..13)+1 \ \land\ d': (0,..13)+1 \ \land\ x'=x \ \land\ y'=y) / 13 / 13.$$
$$x:= \textbf{if } c < n \textbf{ then } c+d \textbf{ else } c.\ \ y:= \textbf{if } c < n+1 \textbf{ then } c+d \textbf{ else } c.$$
$$y<x\le 14 \lor x\le 14<y \qquad \text{Use the Substitution Law twice.}$$
$$= \quad (c': (0,..13)+1 \ \land\ d': (0,..13)+1 \ \land\ x'=x \ \land\ y'=y) / 169.$$
$$(\textbf{if } c < n+1 \textbf{ then } c+d \textbf{ else } c) < (\textbf{if } c < n \textbf{ then } c+d \textbf{ else } c) \le 14$$
$$\lor\ \ (\textbf{if } c < n \textbf{ then } c+d \textbf{ else } c) \le 14 < (\textbf{if } c < n+1 \textbf{ then } c+d \textbf{ else } c)$$
$$= \quad (c': (0,..13)+1 \ \land\ d': (0,..13)+1 \ \land\ x'=x \ \land\ y'=y) / 169.\ \ c=n \land d>14-n$$
$$= \quad \Sigma c'', d'', x'', y''\cdot$$
$$(c'': (0,..13)+1 \ \land\ d'': (0,..13)+1 \ \land\ x''=x \ \land\ y''=y) / 169 \times (c''=n \land d''>14-n)$$
$$= \quad \Sigma d'': 1,..14\cdot (d''>14-n)/169$$
$$= \quad (n-1) / 169$$

The probability that x wins is $(n-1) / 169$. By similar calculations we can find that the probability that y wins is $(14-n) / 169$, and the probability of a tie is $12/13$. For $n<8$, "under $n+1$" beats "under n". For $n\ge 8$, "under n" beats "under $n+1$". So "under 8" beats both "under 7" and "under 9".

Exercise 282 asks: If you repeatedly throw a pair of six-sided dice until they are equal, how long does it take? The program is

$$u'=v' \ \Longleftarrow\ u:= (rand\ 6) + 1.\ \ v:= (rand\ 6) + 1.\ \ \textbf{if } u=v \textbf{ then } ok \textbf{ else } (t:= t+1.\ \ u'=v')$$

Each iteration, with probability $5/6$ we keep going, and with probability $1/6$ we stop. So we offer the hypothesis that (for finite t) the execution time has the distribution

$$(t'\ge t) \times (5/6)^{t'-t} \times 1/6$$

To prove it, let's start with the implementation.

$$u:= (rand\ 6) + 1.\ \ v:= (rand\ 6) + 1. \qquad \text{replace } rand \text{ and}$$
$$\textbf{if } u=v \textbf{ then } t'=t \textbf{ else } (t:= t+1.\ \ (t'\ge t) \times (5/6)^{t'-t} \times 1/6) \qquad \text{Substitution Law}$$
$$= \quad (u': 1,..7 \ \land\ v'=v \ \land\ t'=t)/6.\ \ (u'=u \ \land\ v': 1,..7 \ \land\ t'=t)/6. \qquad \text{replace first .}$$
$$\textbf{if } u=v \textbf{ then } t'=t \textbf{ else } (t'\ge t+1) \times (5/6)^{t'-t-1} / 6 \qquad \text{and simplify}$$
$$= \quad (u', v': 1,..7 \ \land\ t'=t)/36. \qquad \text{replace remaining .}$$
$$\textbf{if } u=v \textbf{ then } t'=t \textbf{ else } (t'\ge t+1) \times (5/6)^{t'-t-1} / 6 \qquad \text{and replace } \textbf{if}$$
$$= \quad \Sigma u'', v'': 1,..7\cdot \Sigma t''\cdot (t''=t)/36 \times \ (\ (u''=v'') \times (t'=t'')$$
$$+ (u''\ne v'') \times (t'\ge t''+1) \times (5/6)^{t'-t''-1} / 6) \qquad \text{sum}$$
$$= \quad (6 \times (t'=t) \ +\ 30 \times (t'\ge t+1) \times (5/6)^{t'-t-1} / 6) / 36 \qquad \text{combine}$$
$$= \quad (t'\ge t) \times (5/6)^{t'-t} \times 1/6$$

which is the distribution we hypothesized, and that completes the proof.

The average value of t' is

$$(t'\ge t) \times (5/6)^{t'-t} \times 1/6.\ \ t \ = \ t+5$$

so on average it takes 5 additional throws of the dice (after the first) to get an equal pair.

End of Random Number Generators

Probability problems are notorious for misleading even professional mathematicians. Informal reasoning to arrive at a probability distribution, as is standard in studies of probability, is essential to forming a reasonable hypothesis. But hypotheses are sometimes wrong. We write the hypothesis as a probabilistic specification, we refine it as a program, and we prove our refinements exactly as we did with boolean specifications. Sometimes wrong hypotheses can be traced to a wrong understanding of the problem. Formalization as a program makes one's understanding clear. Proof shows that a hypothesized probability distribution is correct for the program. Informal arguments are replaced by formal proof.

Probabilistic specifications can also be interpreted as "fuzzy" specifications. For example, $(x'=0)/3 + (x'=1)\times 2/3$ could mean that we will be one-third satisfied if the result x' is 0, two-thirds satisfied if it is 1, and completely unsatisfied if it is anything else.

5.7.1 Information — optional

There is a close connection between information and probability. If a boolean expression has probability p of being true, and you evaluate it, and it turns out to be true, then the amount of information in bits that you have just learned is $info\ p$, defined as

$$info\ p \quad = \quad -\, log\ p$$

where log is the binary (base 2) logarithm. For example, $even\ (rand\ 8)$ has probability $1/2$ of being true. If we evaluate it and find that it is true, we have just learned

$$info\ (1/2) \quad = \quad -\, log\ (1/2) \quad = \quad log\ 2 \quad = \quad 1$$

bit of information; we have learned that the rightmost bit of the random number we were given is 0. If we find that $even\ (rand\ 8)$ is false, then we have learned that $\neg\ even\ (rand\ 8)$ is true, and since it also has probability $1/2$, we have also gained one bit; we have learned that the rightmost bit of the random number is 1. If we test $rand\ 8 = 5$, which has probability $1/8$ of being true, and we find that it is true, we learn

$$info\ (1/8) \quad = \quad -\, log\ (1/8) \quad = \quad log\ 8 \quad = \quad 3$$

bits, which is the entire random number in binary. If we find that $rand\ 8 = 5$ is false, we learn

$$info\ (7/8) \quad = \quad -\, log\ (7/8) \quad = \quad log\ 8 - log\ 7 \quad = \quad 3 - 2.80736 \quad = \quad 0.19264 \text{ approximately}$$

bits; we learn that the random number isn't 5, but it could be any of 7 others. Suppose we test $rand\ 8 < 8$. Since it is certain to be true, there is really no point in making this test; we learn

$$info\ 1 \quad = \quad -\, log\ 1 \quad = \quad -0 \quad = \quad 0$$

When an **if** b **then** P **else** Q occurs within a loop, b is tested repeatedly. Suppose b has probability p of being true. When it is true, we learn $info\ p$ bits, and this happens with probability p. When it is false, we learn $info\ (1-p)$ bits, and this happens with probability $(1-p)$. The average amount of information gained, called the entropy, is

$$entro\ p \quad = \quad p \times info\ p \ + \ (1-p) \times info\ (1-p)$$

For example, $entro\ (1/2) = 1$, and $entro\ (1/8) = entro\ (7/8) = 0.54356$ approximately. Since $entro\ p$ is at its maximum when $p=1/2$, we learn most on average, and make the most efficient use of the test, if its probability is near $1/2$. For example, in the binary search problem of Chapter 4, we could have divided the remaining search interval anywhere, but for the best average execution time, we split it into two parts having equal probabilities of finding the item we seek. And in the fast exponentiation problem, it is better on average to test $even\ y$ rather than $y=0$ if we have a choice.

End of Information

End of Probabilistic Programming

5.8 Functional Programming optional

Most of this book is about a kind of programming that is sometimes called "imperative", which means that a program describes a change of state (or "commands" a computer to change state in a particular way). This section presents an alternative: a program is a function from its input to its output. More generally, a specification is a function from possible inputs to desired outputs, and programs (as always) are implemented specifications. We take away assignment and dependent composition from our programming notations, and we add functions.

To illustrate, we look once again at the list summation problem (Exercise 142). This time, the specification is $\langle L\colon [*rat]\to\Sigma L\rangle$. Assuming Σ is not an implemented operator, we still have some programming to do. We introduce variable n to indicate how much of the list has been summed; initially n is 0 .

$$\Sigma L \;=\; \langle n\colon 0,..\#L+1\to\Sigma\, L\,[n;..\#L]\rangle\; 0$$

It saves some copying to write " $\Sigma L =$ ... " rather than " $\langle L\colon [*rat]\to\Sigma L\rangle =$ ... ", but we must remember the domain of L . At first sight, the domain of n is annoying; it seems to be one occasion when an interval notation that includes both endpoints would be preferable. On second look, it's trying to tell us something useful: the domain is really composed of two parts that must be treated differently.

$$0,..\#L+1 \;=\; (0,..\#L), \#L$$

We divide the function into a selective union

$$\langle n\colon 0,..\#L+1\to\Sigma\, L\,[n;..\#L]\rangle \;=\; \langle n\colon 0,..\#L\to\Sigma\, L\,[n;..\#L]\rangle \mid \langle n\colon \#L\to\Sigma\, L\,[n;..\#L]\rangle$$

and continue with each part separately. In the left part, we have $n<\#L$, and in the right part $n=\#L$.

$$\langle n\colon 0,..\#L\to\Sigma\, L\,[n;..\#L]\rangle \;=\; \langle n\colon 0,..\#L\to Ln + \Sigma\, L\,[n+1;..\#L]\rangle$$

$$\langle n\colon \#L\to\Sigma\, L\,[n;..\#L]\rangle \;=\; \langle n\colon \#L\to 0\rangle$$

This time we copied the domain of n to indicate which part of the selective union is being considered. The one remaining problem is solved by recursion.

$$\Sigma\, L\,[n+1;..\#L] \;=\; \langle n\colon 0,..\#L+1\to\Sigma\, L\,[n;..\#L]\rangle\,(n+1)$$

In place of the selective union we could have used **if then else** ; they are related by the law

$$\langle v\colon A\to x\rangle \mid \langle v\colon B\to y\rangle \;=\; \langle v\colon A, B\to \textbf{if } v\colon A \textbf{ then } x \textbf{ else } y\rangle$$

When we are interested in the execution time rather than the result, we replace the result of each function with its time according to some measure. For example, in the list summation problem, we might decide to charge time 1 for each addition and 0 for everything else. The specification becomes $\langle L\colon [*rat]\to\#L\rangle$, meaning for any list, the execution time is its length. We now must make exactly the same programming steps as before. The first step was to introduce variable n ; we do the same now, but we choose a new result for the new function to indicate its execution time.

$$\#L \;=\; \langle n\colon 0,..\#L+1\to\#L-n\rangle\; 0$$

The second step was to decompose the function into a selective union; we do so again.

$$\langle n\colon 0,..\#L+1\to\#L-n\rangle \;=\; \langle n\colon 0,..\#L\to\#L-n\rangle \mid \langle n\colon \#L\to\#L-n\rangle$$

The left side of the selective union became a function with one addition in it, so our timing function must become a function with a charge of 1 in it. To make the equation correct, the time for the remaining summation must be adjusted.

$$\langle n\colon 0,..\#L\to\#L-n\rangle \;=\; \langle n\colon 0,..\#L\to 1 + \#L-n-1\rangle$$

The right side of the selective union became a function with a constant result; according to our measure, its time must be 0 .

$$\langle n\colon \#L\to\#L-n\rangle \;=\; \langle n\colon \#L\to 0\rangle$$

The remaining problem was solved by a recursive call; the corresponding call solves the remaining time problem.

$$\#L–n–1 \quad = \quad \langle n\colon 0,..\#L+1 \rightarrow \#L–n\rangle\ (n+1)$$

And that completes the proof that execution time (according to this measure) is the length of the list.

In the recursive time measure, we charge nothing for any operation except recursive call, and we charge 1 for that. Let's redo the timing proof with this measure. Again, the time specification is $\langle L\colon [*rat] \rightarrow \#L\rangle$.

$$\#L \ = \ \langle n\colon 0,..\#L+1 \rightarrow \#L–n\rangle\ 0$$

$$\langle n\colon 0,..\#L+1 \rightarrow \#L–n\rangle \quad = \quad \langle n\colon 0,..\#L \rightarrow \#L–n\rangle \mid \langle n\colon \#L \rightarrow \#L–n\rangle$$

$$\langle n\colon 0,..\#L \rightarrow \#L–n\rangle \quad = \quad \langle n\colon 0,..\#L \rightarrow \#L–n\rangle$$

$$\langle n\colon \#L \rightarrow \#L–n\rangle \quad = \quad \langle n\colon \#L \rightarrow 0\rangle$$

$$\#L–n \quad = \quad 1 + \langle n\colon 0,..\#L+1 \rightarrow \#L–n\rangle\ (n+1)$$

5.8.0 Function Refinement

In imperative programming, we can write a nondeterministic specification such as $x'\colon 2, 3, 4$ that allows the result to be any one of several possibilities. In functional programming, a nondeterministic specification is a bunch consisting of more than one element. The specification $2, 3, 4$ allows the result to be any one of those three numbers.

Functional specifications can be classified the same way as imperative specifications, based on the number of satisfactory outputs for each input.

Functional specification S is unsatisfiable for domain element x :	$¢\, Sx\ <\ 1$
Functional specification S is satisfiable for domain element x :	$¢\, Sx\ \ge\ 1$
Functional specification S is deterministic for domain element x :	$¢\, Sx\ \le\ 1$
Functional specification S is nondeterministic for domain element x :	$¢\, Sx\ >\ 1$

Functional specification S is satisfiable for domain element x :	$\exists y\cdot\ y\colon Sx$
Functional specification S is implementable:	$\forall x\cdot\ \exists y\cdot\ y\colon Sx$

(x is quantified over the domain of S , and y is quantified over the range of S .) Implementability can be restated as $\forall x\cdot\ Sx \ne null$.

Consider the problem of searching for an item in a list of integers. Our first attempt at specification might be

$$\langle L\colon [*int] \rightarrow \langle x\colon int \rightarrow §n\colon 0,..\#L\cdot\ Ln = x\rangle\rangle$$

which says that for any list L and item x , we want an index of L where x occurs. If x occurs several times in L , any of its indexes will do. Unfortunately, if x does not occur in L , we are left without any possible result, so this specification is unimplementable. We must decide what we want when x does not occur in L ; let's say any natural that is not an index of L will do.

$$\langle L\colon [*int] \rightarrow \langle x\colon int \rightarrow \mathbf{if}\ x\colon L\ (0,..\#L)\ \mathbf{then}\ §n\colon 0,..\#L\cdot\ Ln = x\ \mathbf{else}\ \#L,..\infty\rangle\rangle$$

This specification is implementable, and often nondeterministic.

Functional refinement is similar to imperative refinement. An imperative specification is a boolean expression, and imperative refinement is reverse implication. Functional specification is a function, and functional refinement is the reverse of the function ordering. Functional specification P (the problem) is refined by functional specification S (the solution) if and only if $S\colon P$. To refine, we can either decrease the choice of result, or increase the domain. Now we have a most annoying notational problem. Typically, we like to write the problem on the left, then the refinement symbol, then the solution on the right; we want to write $S\colon P$ the other way round. Inclusion is antisymmetric, so its symbol should not be symmetric, but unfortunately it is. Let us write :: for “backwards colon”, so that “ P is refined by S ” is written $P\colon\colon S$.

To refine our search specification, we create a linear search program, starting the search with index 0 and increasing the index until either x is found or L is exhausted. First we introduce the index.

$$(\textbf{if } x\text{: } L\ (0,..\#L)\ \textbf{then } §n\text{: } 0,..\#L\cdot\ Ln = x\ \textbf{else } \#L,..\infty)::$$
$$\langle i\text{: } nat \rightarrow \textbf{if } x\text{: } L\ (i,..\#L)\ \textbf{then } §n\text{: } i,..\#L\cdot\ Ln = x\ \textbf{else } \#L,..\infty\rangle\ 0$$

The two sides of this refinement are equal, so we could have written $=$ instead of $::$. We could have been more precise about the domain of i , and then we probably would decompose the function into a selective union, as we did in the previous problem. But this time let's use an **if then else** .

$$(\textbf{if } x\text{: } L\ (i,..\#L)\ \textbf{then } §n\text{: } i,..\#L\cdot\ Ln = x\ \textbf{else } \#L,..\infty)::$$
$$\textbf{if } i = \#L\ \textbf{then } \#L$$
$$\textbf{else if } x = Li\ \textbf{then } i$$
$$\textbf{else } \langle i\text{: } nat \rightarrow \textbf{if } x\text{: } L\ (i,..\#L)\ \textbf{then } §n\text{: } i,..\#L\cdot\ Ln = x\ \textbf{else } \#L,..\infty\rangle\ (i+1)$$

The timing specification, recursive measure, is $\langle L \rightarrow \langle x \rightarrow 0,..\#L+1\rangle\rangle$, which means that the time is less than $\#L+1$. To prove that this is the execution time, we must prove

$$0,..\#L+1::\ \ \langle i\text{: } nat \rightarrow 0,..\#L-i+1\rangle\ 0$$

and

$$0,..\#L-i+1::\quad \textbf{if } i = \#L\ \textbf{then } 0$$
$$\textbf{else if } x = Li\ \textbf{then } 0$$
$$\textbf{else } 1 + \langle i\text{: } nat \rightarrow 0,..\#L-i+1\rangle\ (i+1)$$

As this example illustrates, the steps in a functional refinement are the same as the steps in an imperative refinement for the same problem, including the resolution of nondeterminism and timing. But the notations are different.

End of Function Refinement

Functional and imperative programming are not really competitors; they can be used together. We cannot ignore imperative programming if ever we want to pause, to stop computing for a while, and resume later from the same state. Imperative programming languages all include a functional (expression) sublanguage, so we cannot ignore functional programming either.

At the heart of functional programming we have the Application Axiom

$$\langle v\text{: } D \rightarrow b\rangle\ x\ \ =\ \ (\text{for } v \text{ substitute } x \text{ in } b\ )$$

At the heart of imperative programming we have the Substitution Law

$$x:= e.\ P\ \ =\ \ (\text{for } x \text{ substitute } e \text{ in } P\ )$$

Functional programming and imperative programming differ mainly in the notation they use for substitution.

End of Functional Programming

End of Programming Language

6 Recursive Definition

6.0 Recursive Data Definition

In this section we are concerned with the definition of infinite bunches. Our first example is nat , the natural numbers. It was defined in Chapter 2 using axioms called construction and induction. Now we take a closer look at these axioms.

6.0.0 Construction and Induction

To define nat , we need to say what its elements are. We can start by saying that 0 is an element

$$0: nat$$

and then say that for every element of nat , adding 1 gives an element

$$nat+1: nat$$

These axioms are called the nat construction axioms, and 0 and $nat+1$ are called the nat constructors. Using these axioms, we can "construct" the elements of nat as follows.

	$\top$	by the axiom, $0: nat$
$\Rightarrow$	$0: nat$	add 1 to each side
$\Rightarrow$	$0+1: nat+1$	by arithmetic, $0+1 = 1$; by the axiom, $nat+1: nat$
$\Rightarrow$	$1: nat$	add 1 to each side
$\Rightarrow$	$1+1: nat+1$	by arithmetic, $1+1 = 2$; by the axiom, $nat+1: nat$
$\Rightarrow$	$2: nat$	

and so on.

From the construction axioms we can prove $2: nat$ but we cannot prove $\neg\ -2: nat$. That is why we need the induction axiom. The construction axioms tell us that the natural numbers are in nat , and the induction axiom tells us that nothing else is. Here is the nat induction axiom.

$$0: B\ \land\ B+1: B\ \Rightarrow\ nat: B$$

We have introduced nat as a constant, like $null$ and 0 . It is not a variable, and cannot be instantiated. But B is a variable, to be instantiated at will.

The two construction axioms can be combined into one, and induction can be restated, as follows:

$0, nat+1: nat$	nat construction
$0, B+1: B\ \Rightarrow\ nat: B$	nat induction

There are many bunches satisfying the inclusion $0, B+1: B$, such as: the naturals, the integers, the integers and half-integers, the rationals. Induction says that of all these bunches, nat is the smallest.

We have presented nat construction and nat induction using bunch notation. We now present equivalent axioms using predicate notation. We begin with induction.

In predicate notation, the nat induction axiom can be stated as follows: If $P: nat\to bool$,

$$P0 \land \forall n: nat\cdot Pn \Rightarrow P(n+1)\ \ \Longrightarrow\ \ \forall n: nat\cdot Pn$$

We prove first that the bunch form implies the predicate form.

	$0: B\ \land\ B+1: B\ \Rightarrow\ nat: B$	Let $B = \S n: nat\cdot Pn$. Then $B: nat$,
$\Longrightarrow$	$0: B\ \land (\forall n: nat\cdot n: B\ \Rightarrow\ n+1: B)\ \Rightarrow\ \forall n: nat\cdot n: B$	and $\forall n: nat\cdot (n: B)=Pn$.
$=$	$P0 \land (\forall n: nat\cdot Pn \Rightarrow P(n+1))\ \Rightarrow\ \forall n: nat\cdot Pn$	

The reverse is proven similarly.

$P0 \land (\forall n: nat\cdot Pn \Rightarrow P(n+1)) \;\;\Rightarrow\;\; \forall n: nat\cdot Pn$

For arbitrary bunch B , let $P = \langle n: nat\rightarrow n: B\rangle$. Then again $\forall n: nat\cdot Pn=(n: B)$.

$\Longrightarrow \;\; 0: B \;\land\; (\forall n: nat\cdot n: B \;\Rightarrow\; n+1: B) \;\Rightarrow\; \forall n: nat\cdot n: B$

$= \;\; 0: B \;\land\; (\forall n: nat‘B\cdot n+1: B) \;\Rightarrow\; \forall n: nat\cdot n: B$

$= \;\; 0: B \;\land\; (nat‘B)+1: B \;\Rightarrow\; nat: B$

$\Longrightarrow \;\; 0: B \;\land\; B+1: B \;\Rightarrow\; nat: B$

Therefore the bunch and predicate forms of *nat* induction are equivalent.

The predicate form of *nat* construction can be stated as follows: If $P: nat\rightarrow bool$,

$P0 \land \forall n: nat\cdot Pn \Rightarrow P(n+1) \;\;\Longleftarrow\;\; \forall n: nat\cdot Pn$

This is the same as induction but with the main implication reversed. We prove first that the bunch form implies the predicate form.

$\forall n: nat\cdot Pn$ — domain change using *nat* construction, bunch version

$\Longrightarrow \;\; \forall n: 0, nat+1\cdot Pn$ — axiom about $\forall$

$= \;\; (\forall n: 0\cdot Pn) \land (\forall n: nat+1\cdot Pn)$ — One-Point Law and variable change

$= \;\; P0 \land \forall n: nat\cdot P(n+1)$

$\Longrightarrow \;\; P0 \land \forall n: nat\cdot Pn \Rightarrow P(n+1)$

And now we prove that the predicate form implies the bunch form .

$P0 \land (\forall n: nat\cdot Pn \Rightarrow P(n+1)) \;\;\Leftarrow\;\; \forall n: nat\cdot Pn$ — Let $P = \langle n: nat\rightarrow n: nat\rangle$

$\Longrightarrow \;\; 0: nat \land (\forall n: nat\cdot n: nat \;\Rightarrow\; n+1: nat) \;\Leftarrow\; \forall n: nat\cdot n: nat$

$= \;\; 0: nat \land (\forall n: nat\cdot n+1: nat) \;\Leftarrow\; \top$

$= \;\; 0: nat \;\land\; nat+1: nat$

A corollary is that *nat* can be defined by the single axiom

$P0 \land \forall n: nat\cdot Pn \Rightarrow P(n+1) \;\;=\;\; \forall n: nat\cdot Pn$

There are other predicate versions of induction; here is the usual one again plus three more.

$P0 \land \forall n: nat\cdot Pn \Rightarrow P(n+1) \;\;\Longrightarrow\;\; \forall n: nat\cdot Pn$

$P0 \lor \exists n: nat\cdot \neg Pn \land P(n+1) \;\;\Longleftarrow\;\; \exists n: nat\cdot Pn$

$\forall n: nat\cdot Pn \Rightarrow P(n+1) \;\;\Longrightarrow\;\; \forall n: nat\cdot P0 \Rightarrow Pn$

$\exists n: nat\cdot \neg Pn \land P(n+1) \;\;\Longleftarrow\;\; \exists n: nat\cdot \neg P0 \land Pn$

The first version says that to prove P of all naturals, prove it of 0 , and assuming it of natural n , prove it of $n+1$. In other words, you get to all naturals by starting at 0 and repeatedly adding 1 . The second version is obtained from the first by the duality laws and a renaming. The next is the prettiest; it says that if you can “go” from any natural to the next, then you can “go” from 0 to any natural.

Here are two laws that are consequences of induction.

$\forall n: nat\cdot (\forall m: nat\cdot m<n \Rightarrow Pm) \Rightarrow Pn \;\;\Longrightarrow\;\; \forall n: nat\cdot Pn$

$\exists n: nat\cdot (\forall m: nat\cdot m<n \Rightarrow \neg Pm) \land Pn \;\;\Longleftarrow\;\; \exists n: nat\cdot Pn$

The first is like the first version of induction, except that the base case $P0$ is not explicitly stated, and the step uses the assumption that all previous naturals satisfy P , rather than just the one previous natural. The last one says that if there is a natural with property P then there is a first natural with property P (all previous naturals don't have it).

Proof by induction does not require any special notation or format. For example, Exercise 288 asks us to prove that the square of an odd natural number is $8\times m + 1$ for some natural m. Quantifying over nat,

$$\forall n\cdot \exists m\cdot (2\times n + 1)^2 = 8\times m + 1 \qquad \text{various number laws}$$
$$= \quad \forall n\cdot \exists m\cdot 4\times n\times(n+1) + 1 = 8\times m + 1 \qquad \text{various number laws}$$
$$= \quad \forall n\cdot \exists m\cdot n\times(n+1) = 2\times m \qquad \text{the usual predicate form of induction}$$
$$\Leftarrow \quad (\exists m\cdot 0\times(0+1) = 2\times m) \qquad \text{generalization and}$$
$$\land\ (\forall n\cdot (\exists m\cdot n\times(n+1) = 2\times m) \Rightarrow (\exists l\cdot (n+1)\times(n+2) = 2\times l)) \qquad \text{distribution}$$
$$\Leftarrow \quad 0\times(0+1) = 2\times 0 \qquad \text{arithmetic and}$$
$$\land\ (\forall n, m\cdot n\times(n+1) = 2\times m \Rightarrow (\exists l\cdot (n+1)\times(n+2) = 2\times l)) \qquad \text{generalization}$$
$$\Leftarrow \quad \forall n, m\cdot n\times(n+1) = 2\times m \Rightarrow (n+1)\times(n+2) = 2\times(m+n+1) \qquad \text{various number laws}$$
$$= \quad \top$$

Now that we have an infinite bunch, it is easy to define others. For example, we can define pow to be the powers of 2 either by the equation

$$pow = 2^{nat}$$

or by using the solution quantifier

$$pow = \S p\colon nat\cdot \exists m\colon nat\cdot p = 2^m$$

But let us do it the same way we defined nat. The pow construction axiom is

$$1, 2\times pow\colon pow$$

and the pow induction axiom is

$$1, 2\times B\colon B \Rightarrow pow\colon B$$

Induction is not just for nat. In predicate form, we can define pow with the axiom

$$P1 \land \forall p\colon pow\cdot Pp \Rightarrow P(2\times p) = \forall p\colon pow\cdot Pp$$

We can define the bunch of integers as

$$int = nat, -nat$$

or equivalently we can use the construction and induction axioms

$$0, int+1, int-1\colon int$$
$$0, B+1, B-1\colon B \Rightarrow int\colon B$$

or we can use the axiom

$$P0 \land (\forall i\colon int\cdot Pi \Rightarrow P(i+1)) \land (\forall i\colon int\cdot Pi \Rightarrow P(i-1)) = \forall i\colon int\cdot Pi$$

Whichever we choose as axiom(s), the others are theorems.

Similarly we can define the bunch of rationals as

$$rat = int/(nat+1)$$

or equivalently by the construction and induction axioms

$$1, rat+rat, rat-rat, rat\times rat, rat/(\S r\colon rat\cdot r\neq 0)\colon rat$$
$$1, B+B, B-B, B\times B, B/(\S b\colon B\cdot b\neq 0)\colon B \Rightarrow rat\colon B$$

or with the axiom (quantifying over rat, of course)

$$P1$$
$$\land\ (\forall r, s\cdot Pr \land Ps \Rightarrow P(r+s))$$
$$\land\ (\forall r, s\cdot Pr \land Ps \Rightarrow P(r-s))$$
$$\land\ (\forall r, s\cdot Pr \land Ps \Rightarrow P(r\times s))$$
$$\land\ (\forall r, s\cdot Pr \land Ps \land s\neq 0 \Rightarrow P(r/s))$$
$$= \quad \forall r\cdot Pr$$

As the examples suggest, we can define a bunch by construction and induction axioms using any number of constructors. To end this subsection, we define a bunch using zero constructors. In general, we have one construction axiom per constructor, so there aren't any construction axioms. But there is still an induction axiom. With no constructors, the antecedent becomes trivial and disappears, and we are left with the induction axiom

$null: B$

where $null$ is the bunch being defined. As always, induction says that, apart from elements due to construction axioms, nothing else is in the bunch being defined. This is exactly how we defined $null$ in Chapter 2. The predicate form of $null$ induction is

$\forall x: null \cdot\ P\ x$

End of Construction and Induction

6.0.1 Least Fixed-Points

We have defined nat by a construction axiom and an induction axiom

$0, nat{+}1: nat$ — nat construction
$0, B{+}1: B \ \Rightarrow\ nat: B$ — nat induction

We now prove two similar-looking theorems:

$nat\ =\ 0, nat{+}1$ — nat fixed-point construction
$B = 0, B{+}1\ \Rightarrow\ nat: B$ — nat fixed-point induction

A fixed-point of a function f is an element x of its domain such that f maps x to itself: $x = fx$. A least fixed-point of f is a smallest such x. Fixed-point construction has the form

name = (expression involving *name*)

and so it says that *name* is a fixed-point of the expression on the right. Fixed-point induction tells us that *name* is the smallest bunch satisfying fixed-point construction, and in that sense it is the least fixed-point of the constructor.

We first prove nat fixed-point construction. It is stronger than nat construction, so the proof will also have to use nat induction. Let us start there.

$\top$ — nat induction axiom
$=\ 0, B{+}1: B\ \Rightarrow\ nat: B$ — replace B with $0, nat{+}1$
$\Rightarrow\ 0, (0, nat{+}1){+}1: 0, nat{+}1\ \Rightarrow\ nat: 0, nat{+}1$ — strengthen the antecedent by cancelling the “0”s and “+1”s from the two sides of the first “:”
$\Rightarrow\ 0, nat{+}1: nat\ \Rightarrow\ nat: 0, nat{+}1$ — the antecedent is the nat construction axiom, so we can delete it, and use it again to strengthen the consequent
$=\ nat\ =\ 0, nat{+}1$

We prove nat fixed-point induction just by strengthening the antecedent of nat induction.

In similar fashion we can prove that pow , int , and rat are all least fixed-points of their constructors. In fact, we could have defined nat and each of these bunches as least fixed-points of their constructors. It is quite common to define a bunch of strings by a fixed-point construction axiom called a grammar. For example,

exp = "x", *exp*; "+"; *exp*

In this context, union is usually denoted by | and catenation is usually denoted by nothing. The other axiom, to say that *exp* is the least of the fixed-points, is usually stated informally by saying that only constructed elements are included.

End of Least Fixed-Points

6.0.2 Recursive Data Construction

Recursive construction is a procedure for constructing least fixed-points from constructors. It usually works, but not always. We seek the smallest solution of

$name$ = (expression involving $name$)

Here are the steps of the procedure.

0. Construct a sequence of bunches $name_0$ $name_1$ $name_2$ ... beginning with

$name_0 = null$

and continuing with

$name_{n+1}$ = (expression involving $name_n$)

We can thus construct a bunch $name_n$ for any natural number n .

1. Next, try to find an expression for $name_n$ that may involve n but does not involve $name$.

$name_n$ = (expression involving n but not $name$)

2. Now form a bunch $name_\infty$ by replacing n with ∞ .

$name_\infty$ = (expression involving neither n nor $name$)

3. The bunch $name_\infty$ is usually the least fixed-point of the constructor, but not always, so we must test it. First we test to see if it is a fixed-point.

$name_\infty$ = (expression involving $name_\infty$)

4. Then we test $name_\infty$ to see if it is the least fixed-point.

B = (expression involving B) $\Longrightarrow$ $name_\infty$: B

We illustrate recursive construction on the constructor for pow , which is $1, 2\times pow$.

0. Construct the sequence

$$\begin{aligned}
pow_0 &= null\\
pow_1 &= 1, 2\times pow_0\\
&= 1, 2\times null\\
&= 1, null\\
&= 1\\
pow_2 &= 1, 2\times pow_1\\
&= 1, 2\times 1\\
&= 1, 2\\
pow_3 &= 1, 2\times pow_2\\
&= 1, 2\times(1, 2)\\
&= 1, 2, 4
\end{aligned}$$

The first bunch pow_0 tells us all the elements of the bunch pow that we know without looking at its constructor. In general, pow_n represents our knowledge of pow after n uses of its constructor.

1. Perhaps now we can guess the general member of this sequence

$pow_n \;=\; 2^{0,..n}$

We could prove this by *nat* induction, but it is not really necessary. The proof would only tell us about pow_n for n: *nat* and we want pow_∞ . Besides, we will test our final result.

2. Now that we can express pow_n , we can define pow_∞ as

$pow_\infty \;=\; 2^{0,..\infty}$
$\;=\; 2^{nat}$

and we have found a likely candidate for the least fixed-point of the *pow* constructor.

3. We must test pow_∞ to see if it is a fixed-point.

	$2^{nat} = 1, 2\times 2^{nat}$	
$=$	$2^{nat} = 2^0, 2^1\times 2^{nat}$	
$=$	$2^{nat} = 2^0, 2^{1+nat}$	
$=$	$2^{nat} = 2^{0,\,1+nat}$	
$\Leftarrow$	$nat = 0, nat+1$	*nat* fixed-point construction
$=$	$\top$	

4. We must test pow_∞ to see if it is the least fixed-point.

	$2^{nat}: B$	
$=$	$\forall n: nat\cdot 2^n: B$	use the predicate form of *nat* induction
$\Leftarrow$	$2^0: B \;\wedge\; \forall n: nat\cdot 2^n: B \;\Rightarrow\; 2^{n+1}: B$	change variable
$=$	$1: B \;\wedge\; \forall m: 2^{nat}\cdot m: B \;\Rightarrow\; 2\times m: B$	increase domain
$\Leftarrow$	$1: B \;\wedge\; \forall m: nat\cdot m: B \;\Rightarrow\; 2\times m: B$	Domain Change Law
$=$	$1: B \;\wedge\; \forall m: nat`B\cdot 2\times m: B$	increase domain
$\Leftarrow$	$1: B \;\wedge\; \forall m: B\cdot 2\times m: B$	
$=$	$1: B \;\wedge\; 2\times B: B$	
$\Leftarrow$	$B = 1, 2\times B$	

Since 2^{nat} is the least fixed-point of the *pow* constructor, we conclude $pow = 2^{nat}$.

In step 0, we start with $name_0 = null$, which is usually the best starting bunch for finding a smallest solution. But occasionally that starting bunch fails and some other starting bunch succeeds in producing a solution to the given fixed-point equation.

In step 2, from $name_n$ we obtain a candidate $name_\infty$ for a fixed-point of a constructor by replacing n with ∞ . This substitution is simple to perform, and the resulting candidate is usually satisfactory. But the result is sensitive to the way $name_n$ is expressed. From two expressions for $name_n$ that are equal for all finite n , we may obtain expressions for $name_\infty$ that are unequal. Another candidate, one that is not sensitive to the way $name_n$ is expressed, is

$\S x\cdot LIM\ n\cdot x: name_n$

But this bunch is sensitive to the choice of domain of x (the domain of n has to be *nat*). Finding a limit is harder than making a substitution, and the result is still not guaranteed to produce a fixed-point. We could define a property, called "continuity", which, together with monotonicity, is sufficient to guarantee that the limit is the least fixed-point, but we leave the subject to other books.

End of Recursive Data Construction

Whenever we add axioms, we must be careful to remain consistent with the theory we already have. A badly chosen axiom can cause inconsistency. Here is an example. Suppose we make

$$bad \;=\; \S n\colon nat\cdot \neg\, n\colon bad$$

an axiom. Thus *bad* is defined as the bunch of all naturals that are not in *bad* . From this axiom we find

$$0\colon bad$$
$$=\; 0\colon \S n\colon nat\cdot \neg\, n\colon bad$$
$$=\; \neg\, 0\colon bad$$

is a theorem. From the Completion Rule we find that $0\colon bad = \neg\, 0\colon bad$ is also an antitheorem. To avoid the inconsistency, we must withdraw this axiom.

Sometimes recursive construction does not produce any answer. For example, the fixed-point equation of the previous paragraph results in the sequence of bunches

$$bad_0 \;=\; null$$
$$bad_1 \;=\; nat$$
$$bad_2 \;=\; null$$

and so on, alternating between *null* and *nat* . We cannot say what bad_∞ is because we cannot say whether ∞ is even or odd. Even the Limit Axiom tells us nothing. We should not blame recursive construction for failing to find a fixed-point when there is none. However, it sometimes fails to find a fixed-point when there is one (see Exercise 314).

End of Recursive Data Definition

6.1 Recursive Program Definition

Programs, and more generally, specifications, can be defined by axioms just as data can. For our first example, let x and y be integer variables. The name *zap* is introduced, and the fixed-point equation

$$zap \;=\; \textbf{if}\ x{=}0\ \textbf{then}\ y{:=}\ 0\ \textbf{else}\ (x{:=}\ x{-}1.\ \ t{:=}\ t{+}1.\ \ zap)$$

is given as an axiom. The right side of the equation is the constructor. Here are six solutions to this equation.

(a) $x{\ge}0 \;\Rightarrow\; x'{=}y'{=}0 \land t' = t{+}x$

(b) $\textbf{if}\ x{\ge}0\ \textbf{then}\ x'{=}y'{=}0 \land t' = t{+}x\ \textbf{else}\ t'{=}\infty$

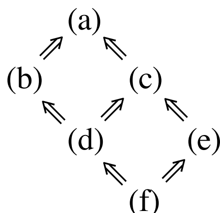
(c) $x'{=}y'{=}0 \land (x{\ge}0 \Rightarrow t' = t{+}x)$

(d) $x'{=}y'{=}0 \land \textbf{if}\ x{\ge}0\ \textbf{then}\ t' = t{+}x\ \textbf{else}\ t'{=}\infty$

(e) $x'{=}y'{=}0 \land t' = t{+}x$

(f) $x{\ge}0 \land x'{=}y'{=}0 \land t' = t{+}x$

Solution (a) is the weakest and solution (f) is the strongest, although the solutions are not totally ordered. We can express their order by the following picture.



Solutions (e) and (f) are so strong that they are unimplementable. Solution (d) is implementable, and since it is also deterministic, it is a strongest implementable solution.

From the fixed-point equation defining zap , we cannot say that zap is equal to a particular one of the solutions. But we can say this: it refines the weakest solution

$$x{\ge}0 \Rightarrow x'{=}y'{=}0 \land t' = t{+}x \quad \Leftarrow \quad zap$$

so we can use it to solve problems. And it is refined by its constructor

$$zap \quad \Leftarrow \quad \textbf{if } x{=}0 \textbf{ then } y{:=}\ 0 \textbf{ else } (x{:=}\ x{-}1.\ \ t{:=}\ t{+}1.\ \ zap)$$

so we can execute it. For all practical purposes, that is all we need.

6.1.0 Recursive Program Construction

Recursive program construction is similar to recursive data construction, and serves a similar purpose. We illustrate the procedure using the example zap . We start with zap_0 describing the computation as well as we can without looking at the definition of zap . Of course, if we don't look at the definition, we have no idea what computation zap is describing, so let us start with a specification that is satisfied by every computation.

$$zap_0 = \top$$

We obtain the next description of zap by substituting zap_0 for zap in the constructor, and so on.

$$\begin{aligned} zap_1 &= \textbf{if } x{=}0 \textbf{ then } y{:=}\ 0 \textbf{ else } (x{:=}\ x{-}1.\ \ t{:=}\ t{+}1.\ \ zap_0) \\ &= x{=}0 \ \Rightarrow\ x'{=}y'{=}0 \land t'{=}t \\ zap_2 &= \textbf{if } x{=}0 \textbf{ then } y{:=}\ 0 \textbf{ else } (x{:=}\ x{-}1.\ \ t{:=}\ t{+}1.\ \ zap_1) \\ &= 0{\le}x{<}2 \ \Rightarrow\ x'{=}y'{=}0 \land t' = t{+}x \end{aligned}$$

In general, zap_n describes the computation as well as possible after n uses of the constructor. We can now guess (and prove using nat induction if we want)

$$zap_n = 0{\le}x{<}n \ \Rightarrow\ x'{=}y'{=}0 \land t' = t{+}x$$

The next step is to replace n with ∞ .

$$zap_\infty = 0{\le}x{<}\infty \ \Rightarrow\ x'{=}y'{=}0 \land t' = t{+}x$$

Finally, we must test the result to see if it satisfies the axiom.

$$\begin{aligned} & \text{(right side of equation with } zap_\infty \text{ for } zap\text{)} \\ = \ & \textbf{if } x{=}0 \textbf{ then } y{:=}\ 0 \textbf{ else } (x{:=}\ x{-}1.\ \ t{:=}\ t{+}1.\ \ 0{\le}x \ \Rightarrow\ x'{=}y'{=}0 \land t' = t{+}x) \\ = \ & \textbf{if } x{=}0 \textbf{ then } x'{=}y'{=}0 \land t'{=}t \textbf{ else } 0{\le}x{-}1 \ \Rightarrow\ x'{=}y'{=}0 \land t' = t{+}x \\ = \ & 0{\le}x \ \Rightarrow\ x'{=}y'{=}0 \land t' = t{+}x \\ = \ & \text{(left side of equation with } zap_\infty \text{ for } zap\text{)} \end{aligned}$$

It satisfies the fixed-point equation, and in fact it is the weakest fixed-point.

If we are not considering time, then $\top$ is all we can say about an unknown computation, and we start our recursive construction there. With time, we can say more than just $\top$; we can say that time does not decrease. Starting with $t' \ge t$ we can construct a stronger fixed-point.

$$\begin{aligned} zap_0 &= t' \ge t \\ zap_1 &= \textbf{if } x{=}0 \textbf{ then } y{:=}\ 0 \textbf{ else } (x{:=}\ x{-}1.\ \ t{:=}\ t{+}1.\ \ zap_0) \\ &= \textbf{if } x{=}0 \textbf{ then } x'{=}y'{=}0 \land t'{=}t \textbf{ else } t' \ge t{+}1 \\ zap_2 &= \textbf{if } x{=}0 \textbf{ then } y{:=}\ 0 \textbf{ else } (x{:=}\ x{-}1.\ \ t{:=}\ t{+}1.\ \ zap_1) \\ &= \textbf{if } x{=}0 \textbf{ then } x'{=}y'{=}0 \land t'{=}t \textbf{ else if } x{=}1 \textbf{ then } x'{=}y'{=}0 \land t'{=}t{+}1 \textbf{ else } t' \ge t{+}2 \\ &= \textbf{if } 0{\le}x{<}2 \textbf{ then } x'{=}y'{=}0 \land t' = t{+}x \textbf{ else } t' \ge t{+}2 \end{aligned}$$

In general, zap_n describes what we know up to time n . We can now guess (and prove using nat induction if we want)

$$zap_n = \textbf{if } 0{\le}x{<}n \textbf{ then } x'{=}y'{=}0 \land t'{=}t{+}x \textbf{ else } t' \ge t{+}n$$

We replace n with ∞

$$zap_\infty \;=\; \textbf{if } 0{\le}x \textbf{ then } x'{=}y'{=}0 \land t'{=}t{+}x \textbf{ else } t'{=}\infty$$

and test the result

$$\begin{aligned}
& \text{(right side of equation with } zap_\infty \text{ for } zap\text{)}\\
=\;& \textbf{if } x{=}0 \textbf{ then } y{:=}\,0 \textbf{ else } (x{:=}\,x{-}1.\;\; t{:=}\,t{+}1.\;\; \textbf{if } 0{\le}x \textbf{ then } x'{=}y'{=}0 \land t'{=}t{+}x \textbf{ else } t'{=}\infty)\\
=\;& \textbf{if } x{=}0 \textbf{ then } x'{=}y'{=}0 \land t'{=}t \textbf{ else if } 0{\le}x{-}1 \textbf{ then } x'{=}y'{=}0 \land t'{=}t{+}x \textbf{ else } t'{=}\infty\\
=\;& \textbf{if } 0{\le}x \textbf{ then } x'{=}y'{=}0 \land t' = t{+}x \textbf{ else } t'{=}\infty\\
=\;& \text{(left side of equation with } zap_\infty \text{ for } zap\text{)}
\end{aligned}$$

Beginning our recursive construction with $t' \ge t$, we have constructed a stronger but still implementable fixed-point. In this example, if we begin our recursive construction with $\bot$ we obtain the strongest fixed-point, which is unimplementable.

We obtained a candidate zap_∞ for a fixed-point by replacing n with ∞. An alternative candidate is $LIM\ n\cdot\ zap_n$. In this example, the two candidates are equal, but in other examples the two ways of forming a candidate may give different results.

End of Recursive Program Construction

6.1.1 Loop Definition

Loops can be defined by construction and induction. The axioms for the **while**-loop are

$$\begin{aligned}
& t'{\ge}t \;\Leftarrow\; \textbf{while } b \textbf{ do } P\\
& \textbf{if } b \textbf{ then } (P.\;\; t{:=}\,t{+}1.\;\; \textbf{while } b \textbf{ do } P) \textbf{ else } ok \;\Leftarrow\; \textbf{while } b \textbf{ do } P\\
& \qquad \forall\sigma, \sigma'\cdot\ (t'{\ge}t \land (\textbf{if } b \textbf{ then } (P.\;\; t{:=}\,t{+}1.\;\; W) \textbf{ else } ok) \;\Leftarrow\; W)\\
& \Rightarrow\;\; \forall\sigma, \sigma'\cdot\ (\textbf{while } b \textbf{ do } P \;\Leftarrow\; W)
\end{aligned}$$

Recursive timing has been included, but this can be changed to any other timing policy. These three axioms are closely analogous to the axioms

$$\begin{aligned}
& 0\colon nat\\
& nat{+}1\colon nat\\
& 0, B{+}1\colon B \;\Rightarrow\; nat\colon B
\end{aligned}$$

that define nat. The first **while**-loop axiom is a base case saying that at least time does not decrease. The second construction axiom takes a single step, saying that **while** b **do** P refines (implements) its first unrolling; then by Stepwise Refinement we can prove it refines any of its unrollings. The last axiom, induction, says that it is the weakest specification that satisfies the first two axioms.

From these axioms we can prove theorems called fixed-point construction and fixed-point induction. For the **while**-loop they are

$$\begin{aligned}
& \textbf{while } b \textbf{ do } P \;=\; t'{\ge}t \land (\textbf{if } b \textbf{ then } (P.\;\; t{:=}\,t{+}1.\;\; \textbf{while } b \textbf{ do } P) \textbf{ else } ok)\\
& \qquad \forall\sigma, \sigma'\cdot\ (W \;=\; t'{\ge}t \land (\textbf{if } b \textbf{ then } (P.\;\; t{:=}\,t{+}1.\;\; W) \textbf{ else } ok))\\
& \Rightarrow\;\; \forall\sigma, \sigma'\cdot\ (\textbf{while } b \textbf{ do } P \;\Leftarrow\; W)
\end{aligned}$$

This account differs from that presented in Chapter 5; we have gained some theorems, and also lost some theorems. For example, from this definition, we cannot prove

$$x'{\ge}x \;\Leftarrow\; \textbf{while } b \textbf{ do } x'{\ge}x$$

which was easily proved according to Chapter 5.

End of Loop Definition

End of Recursive Program Definition

End of Recursive Definition

7 Theory Design and Implementation

Programmers use the formalisms, abstractions, theories, and structures that have been created for them by the designers and implementers of their programming languages. With every program they write, with every name they introduce, programmers create new formalisms, abstractions, theories, and structures. To make their creations as elegant and useful as possible, programmers should be aware of their role as theory designers and implementers, as well as theory users.

The stack, the queue, and the tree are standard data structures used frequently in programming. It is not the purpose of the present chapter to show their usefulness in applications; we leave that to books devoted to data structures. They are presented here as case studies in theory design and implementation. Each of these data structures contains items of some sort. For example, we can have stacks of integers, stacks of lists of booleans, even stacks of stacks. In this chapter, X is the bunch (or type) of items in a data structure.

7.0 Data Theories

7.0.0 Data-Stack Theory

The stack is a useful data structure for the implementation of programming languages. Its distinguishing feature is that, at any time, the item to be inspected or deleted next is the newest remaining item. It is the structure with the motto: the last one in is the first one out.

We introduce the syntax *stack* , *empty* , *push* , *pop* , and *top* . Informally, they mean the following.

stack a bunch consisting of all stacks of items of type X
empty a stack containing no items (an element of bunch *stack*)
push a function that, given a stack and an item, gives back the stack containing the same items plus the one new item
pop a function that, given a stack, gives back the stack minus the newest remaining item
top a function that, given a stack, gives back the newest remaining item

Here are the first four axioms.

$empty: stack$
$push: stack \to X \to stack$
$pop: stack \to stack$
$top: stack \to X$

We want *empty* and *push* to be *stack* constructors. We want a stack obtained by *pop* to be one that was constructed from *empty* and *push* , so we do not need *pop* to be a constructor. A construction axiom can be written in either of the following two ways:

$empty, push\ stack\ X: stack$
$P\ empty \land \forall s: stack\cdot \forall x: X\cdot Ps \Rightarrow P(push\ s\ x) \quad \Longleftarrow \quad \forall s: stack\cdot Ps$

where *push* is allowed to distribute over bunch union, and $P: stack \to bool$. To exclude anything else from being a stack requires an induction axiom, which can be written in many ways; here are two:

$empty, push\ B\ X: B \ \Rightarrow\ stack: B$
$P\ empty \land \forall s: stack\cdot \forall x: X\cdot Ps \Rightarrow P(push\ s\ x) \quad \Longrightarrow \quad \forall s: stack\cdot Ps$

According to the axioms we have so far, it is possible that all stacks are equal. To say that the constructors always construct different stacks requires two more axioms. Let $s, t: stack$ and

x, y: X ; then

$push\ s\ x \neq empty$

$push\ s\ x = push\ t\ y \quad = \quad s{=}t \wedge x{=}y$

And finally, two axioms are needed to say that stacks behave in “last in, first out” fashion.

$pop\ (push\ s\ x) = s$

$top\ (push\ s\ x) = x$

And that completes the data-stack axioms.

End of Data-Stack Theory

Data-stack theory allows us to declare as many stack variables as we want and to use them in expressions according to the axioms. We can declare variables a and b of type *stack* , and then write the assignments $a{:=}\ empty$ and $b{:=}\ push\ a\ 2$.

7.0.1 Data-Stack Implementation

If you need a stack and stacks are not provided in your programming language, you will have to build your stack using the data structures that are provided. Suppose that lists and functions are implemented. Then we can implement a stack of integers by the following definitions.

$stack \ = \ [*int]$

$empty \ = \ [nil]$

$push \ = \ \langle s\colon stack \to \langle x\colon int \to s{+}[x]\rangle\rangle$

$pop \ = \ \langle s\colon stack \to \textbf{if } s{=}empty \textbf{ then } empty \textbf{ else } s\ [0;..\#s{-}1]\rangle$

$top \ = \ \langle s\colon stack \to \textbf{if } s{=}empty \textbf{ then } 0 \textbf{ else } s\ (\#s{-}1)\rangle$

To prove that a theory is implemented, we prove

(the axioms of the theory) $\Leftarrow$ (the definitions of the implementation)

In other words, the definitions must satisfy the axioms. According to a distributive law, this can be done one axiom at a time. For example, the last axiom becomes

$top\ (push\ s\ x) = x$ — replace *push*

$=\ top\ (\langle s\colon stack \to \langle x\colon int \to s{+}[x]\rangle\rangle\ s\ x) = x$ — apply function

$=\ top\ (s{+}[x]) = x$ — replace *top*

$=\ \langle s\colon stack \to \textbf{if } s{=}empty \textbf{ then } 0 \textbf{ else } s\ (\#s{-}1)\rangle\ (s{+}[x]) = x$ — apply function and replace *empty*

$=\ (\textbf{if } s{+}[x]{=}[nil] \textbf{ then } 0 \textbf{ else } (s{+}[x])\ (\#(s{+}[x]){-}1)) = x$ — simplify the **if** and the index

$=\ (s{+}[x])\ (\#s) = x$ — index the list

$=\ x = x$ — reflexive law

$=\ \top$

End of Data-Stack Implementation

Is stack theory consistent? Since we implemented it using list theory, we know that if list theory is consistent, so is stack theory. Is stack theory complete? To show that a boolean expression is unclassified, we must implement stacks twice, making the expression a theorem in one implementation, and an antitheorem in the other. The expressions

$pop\ empty = empty$

$top\ empty = 0$

are theorems in our implementation, but we can alter the implementation as follows

$pop \ = \ \langle s\colon stack \to \textbf{if } s{=}empty \textbf{ then } push\ empty\ 0 \textbf{ else } s\ [0;..\#s{-}1]\rangle$

$top \ = \ \langle s\colon stack \to \textbf{if } s{=}empty \textbf{ then } 1 \textbf{ else } s\ (\#s{-}1)\rangle$

to make them antitheorems. So stack theory is incomplete.

Stack theory specifies the properties of stacks. A person who implements stacks must ensure that all these properties are provided. A person who uses stacks must ensure that only these properties are relied upon. This point deserves emphasis: a theory is a contract between two parties, an implementer and a user (they may be one person with two hats, or two corporations). It makes clear what each party's obligations are to the other, and what each can expect from the other. If something goes wrong, it makes clear who is at fault. A theory makes it possible for each side to modify their part of a program without knowing how the other part is written. This is an essential principle in the construction of large-scale software. In our small example, the stack user must not use $pop\ empty = empty$ even though the stack implementer has provided it; if the user wants it, it should be added to the theory.

7.0.2 Simple Data-Stack Theory

In the data-stack theory just presented, we have axioms $empty: stack$ and $pop: stack \rightarrow stack$; from them we can prove $pop\ empty: stack$. In other words, popping the empty stack gives a stack, though we do not know which one. An implementer is obliged to give a stack for $pop\ empty$, though it does not matter which one. If we never want to pop an empty stack, then the theory is too strong. We should weaken the axiom $pop: stack \rightarrow stack$ and remove the implementer's obligation to provide something that is not wanted. The weaker axiom

$$s \neq empty \;\Rightarrow\; pop\ s: stack$$

says that popping a nonempty stack yields a stack, but it is implied by the remaining axioms and so is unnecessary. Similarly from $empty: stack$ and $top: stack \rightarrow X$ we can prove $top\ empty: X$; deleting the axiom $top: stack \rightarrow X$ removes an implementer's obligation to provide an unwanted result for $top\ empty$.

We may decide that we have no need to prove anything about all stacks, and can do without $stack$ induction. After a little thought, we may realize that we never need an empty stack, nor to test if a stack is empty. We can always work on top of a given (possibly non-empty) stack, and in most uses we are required to do so, leaving the stack as we found it. We can delete the axiom $empty: stack$ and all mention of $empty$. We must replace this axiom with the weaker axiom $stack \neq null$ so that we can still declare variables of type $stack$. If we want to test whether a stack is empty, we can begin by pushing some special value, one that will not be pushed again, onto the stack; the empty test is then a test whether the top is the special value.

For most purposes, it is sufficient to be able to push items onto a stack, pop items off, and look at the top item. The theory we need is considerably simpler than the one presented previously. Our simpler data-stack theory introduces the names $stack$, $push$, pop , and top with the following four axioms: Let $s: stack$ and $x: X$; then

$$stack \neq null$$
$$push\ s\ x: stack$$
$$pop\ (push\ s\ x) = s$$
$$top\ (push\ s\ x) = x$$

End of Simple Data-Stack Theory

For the purpose of studying stacks, as a mathematical activity, we want the strongest axioms possible so that we can prove as much as possible. As an engineering activity, theory design is the art of excluding all unwanted implementations while allowing all the others. It is counter-productive to design a stronger theory than necessary; it makes implementation harder, and it makes theory extension harder.

7.0.3 Data-Queue Theory

The queue data structure, also known as a buffer, is useful in simulations and scheduling. Its distinguishing feature is that, at any time, the item to be inspected or deleted next is the oldest remaining item. It is the structure with the motto: the first one in is the first one out.

We introduce the syntax *queue* , *emptyq* , *join* , *leave* , and *front* with the following informal meaning:

queue	a bunch consisting of all queues of items of type X
emptyq	a queue containing no items (an element of bunch *queue*)
join	a function that, given a queue and an item, gives back the queue containing the same items plus the one new item
leave	a function that, given a queue, gives back the queue minus the oldest remaining item
front	a function that, given a queue, gives back the oldest remaining item

The same kinds of considerations that went into the design of stack theory also guide the design of queue theory. Let q, r: *queue* and x, y: X . We certainly want the construction axioms

$$emptyq\text{: } queue$$
$$join\ q\ x\text{: } queue$$

If we want to prove things about the domain of *join* , then we must replace the second construction axiom by the stronger axiom

$$join\text{: } queue \to X \to queue$$

To say that the constructors construct distinct queues, with no repetitions, we need

$$join\ q\ x \neq emptyq$$
$$join\ q\ x = join\ r\ y \quad = \quad q{=}r \land x{=}y$$

We want a queue obtained by *leave* to be one that was constructed from *emptyq* and *join* , so we do not need

$$leave\ q\text{: } queue$$

for construction, and we don't want to oblige an implementer to provide a representation for *leave emptyq* , so perhaps we will omit that one. We do want to say

$$q{\neq}emptyq \ \Rightarrow\ leave\ q\text{: } queue$$

And similarly, we want

$$q{\neq}emptyq \ \Rightarrow\ front\ q\text{: } X$$

If we want to prove something about all queues, we need *queue* induction:

$$emptyq, join\ B\ X\text{: } B \ \Rightarrow\ queue\text{: } B$$

And finally, we need to give queues their “first in, first out” character:

$$leave\ (join\ emptyq\ x) \ = \ emptyq$$
$$q{\neq}emptyq \ \Rightarrow\ leave\ (join\ q\ x) = join\ (leave\ q)\ x$$
$$front\ (join\ emptyq\ x) \ = \ x$$
$$q{\neq}emptyq \ \Rightarrow\ front\ (join\ q\ x) = front\ \ q$$

If we have decided to keep the *queue* induction axiom, we can throw away the two earlier axioms

$$q{\neq}emptyq \ \Rightarrow\ leave\ q\text{: } queue$$
$$q{\neq}emptyq \ \Rightarrow\ front\ q\text{: } X$$

since they can now be proven.

End of Data-Queue Theory

After data-stack implementation, data-queue implementation raises no new issues, so we leave it as Exercise 340.

7.0.4 Data-Tree Theory

We introduce the syntax

tree	a bunch consisting of all finite binary trees of items of type X
emptree	a tree containing no items (an element of bunch *tree*)
graft	a function that, given two trees and an item, gives back the tree with the item at the root and the two given trees as left and right subtree
left	a function that, given a tree, gives back its left subtree
right	a function that, given a tree, gives back its right subtree
root	a function that, given a tree, gives back its root item

For the purpose of studying trees, we want a strong theory. Let t, u, v, w: *tree* and x, y: X .

$emptree: tree$
$graft: tree \to X \to tree \to tree$
$emptree, graft\ B\ X\ B: B \ \Rightarrow\ tree: B$
$graft\ t\ x\ u \neq emptree$
$graft\ t\ x\ u = graft\ v\ y\ w \ = \ t{=}v \wedge x{=}y \wedge u{=}w$
$left\ (graft\ t\ x\ u) = t$
$root\ (graft\ t\ x\ u) = x$
$right\ (graft\ t\ x\ u) = u$

where, in the construction axiom, *graft* is allowed to distribute over bunch union.

For most programming purposes, the following simpler, weaker theory is sufficient.

$tree \neq null$
$graft\ t\ x\ u: tree$
$left\ (graft\ t\ x\ u) = t$
$root\ (graft\ t\ x\ u) = x$
$right\ (graft\ t\ x\ u) = u$

As with stacks, we don't really need to be given an empty tree. As long as we are given some tree, we can build a tree with a distinguished root that serves the same purpose. And we probably don't need *tree* induction.

End of Data-Tree Theory

7.0.5 Data-Tree Implementation

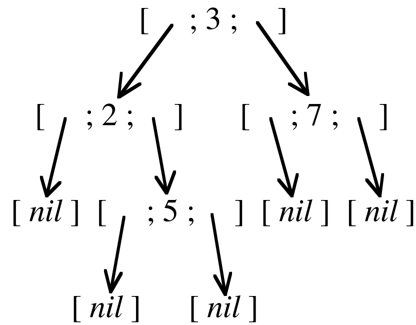
Suppose lists and recursive data definition are implemented. Then we can implement a tree of integers by the following definitions:

$tree\ =\ emptree, graft\ tree\ int\ tree$
$emptree\ =\ [nil]$
$graft\ =\ \langle t: tree \to \langle x: int \to \langle u: tree \to [t; x; u] \rangle\rangle\rangle$
$left\ =\ \langle t: tree \to t\ 0 \rangle$
$right\ =\ \langle t: tree \to t\ 2 \rangle$
$root\ =\ \langle t: tree \to t\ 1 \rangle$

The procedure *graft* makes a list of three items; two of those items are lists themselves. A reasonable implementation strategy for lists is to allocate a small space, one capable of holding an integer or data address, for each item. If an item is an integer, it is put in its place; if an item is a list, it is put somewhere else and a pointer to it (data address) is put in its place. In this implementation of lists, pointers are provided automatically when needed. For example, the tree

$[[[nil]; 2; [[nil]; 5; [nil]]]; 3; [[nil]; 7; [nil]]]$

looks like



Here is another implementation of data-trees.

$tree = emptree, graft\ tree\ int\ tree$
$emptree = 0$
$graft = \langle t: tree \rightarrow \langle x: int \rightarrow \langle u: tree \rightarrow (\text{"left"} \rightarrow t \mid \text{"root"} \rightarrow x \mid \text{"right"} \rightarrow u)\rangle\rangle\rangle$
$left = \langle t: tree \rightarrow t\ \text{"left"}\rangle$
$right = \langle t: tree \rightarrow t\ \text{"right"}\rangle$
$root = \langle t: tree \rightarrow t\ \text{"root"}\rangle$

With this implementation, a tree value looks like this:

```
  "left" →   ("left" → 0
             | "root" → 2
             | "right" →  ("left" → 0
                          | "root" → 5
                          | "right" → 0 ) )
| "root" → 3
| "right" →  ("left" → 0
             | "root" → 7
             | "right" → 0 )
```

If the implementation you have available does not include recursive data definition, you will have to build the pointer structure yourself. For example, in C you can code the implementation of binary trees as follows:

```
struct tree { struct tree *left;  int root;  struct tree *right; };
struct tree *emptree = NULL;
struct tree *graft (struct tree *t, int x, struct tree *u)
    {   struct tree *g;  g = malloc (sizeof(struct tree));
        (*g).left = t;  (*g).root = x;  (*g).right = u;
        return g;
    }
struct tree *left (struct tree *t) { return (*t).left; }
int root (struct tree *t) { return (*t).root; }
struct tree *right (struct tree *t) { return (*t).right; }
```

As you can see, the C code is clumsy. It is not a good idea to apply Program Theory directly to the C code. The use of pointers (data addresses) when recursive data definition is unimplemented is just like the use of **go to** (program addresses) when recursive program definition is unimplemented or implemented badly.

End of Data-Tree Implementation

End of Data Theories

A data theory creates a new type, or value space, or perhaps an extension of an old type. A program theory creates new programs, or rather, new specifications that become programs when the theory is implemented. These two styles of theory correspond to two styles of programming: functional and imperative.

7.1 Program Theories

In program theories, the state is divided into two kinds of variables: the user's variables and the implementer's variables. A user of the theory enjoys full access to the user's variables, but cannot directly access (see or change) the implementer's variables. A user gets access to the implementer's variables only through the theory. On the other side, an implementer of the theory enjoys full access to the implementer's variables, but cannot directly access (see or change) the user's variables. An implementer gets access to the user's variables only through the theory. Some programming languages have a “module” or “object” construct exactly for this purpose. In other languages we just forbid the use of the wrong variables on each side of the boundary.

If we need only one stack or one queue or one tree, we can obtain an economy of expression and of execution by leaving it implicit. There is no need to say which stack to push onto if there is only one, and similarly for the other operations and data structures. Each of the program theories we present will provide only one of its type of data structure to the user, but they can be generalized by adding an extra parameter to each operation.

7.1.0 Program-Stack Theory

The simplest version of program-stack theory introduces three names: *push* (a procedure with parameter of type X), *pop* (a program), and *top* (of type X). In this theory, $push\ 3$ is a program (assuming $3: X$); it changes the state. Following this program, before any other pushes and pops, *print top* will print 3. The following two axioms are sufficient.

$$top'=x \Leftarrow push\ x$$
$$ok \Leftarrow push\ x.\ pop$$

where $x: X$.

The second axiom says that a pop undoes a push. In fact, it says that any natural number of pushes are undone by the same number of pops.

	ok	use second axiom
$\Leftarrow$	$push\ x.\ pop$	ok is identity for dependent composition
$=$	$push\ x.\ ok.\ pop$	Refinement by Steps reusing the axiom
$\Leftarrow$	$push\ x.\ push\ y.\ pop.\ pop$	

We can prove things like

$$top'=x \Leftarrow push\ x.\ push\ y.\ push\ z.\ pop.\ pop$$

which say that when we push something onto the stack, we find it there later at the appropriate time. That is all we really want.

End of Program-Stack Theory

7.1.1 Program-Stack Implementation

To implement program-stack theory, we introduce an implementer's variable $s: [*X]$ and define

$$push = \langle x: X \rightarrow s:= s^+[x] \rangle$$
$$pop = s:= s\ [0;..\#s-1]$$
$$top = s\ (\#s-1)$$

And, of course, we must show that these definitions satisfy the axioms. We'll do the first axiom, and leave the other as Exercise 342.

$(\ top'{=}x \ \Leftarrow\ push\ x\ )$ — use definition of *push* and *top*

$=\ (\ s'(\#s'-1){=}x \ \Leftarrow\ s{:=}\ s{+}[x]\ )$ — List Theory

$=\ \top$

End of Program-Stack Implementation

7.1.2 Fancy Program-Stack Theory

The program-stack theory just presented corresponds to the simpler data-stack theory presented earlier. A slightly fancier program-stack theory introduces two more names: *mkempty* (a program to make the stack empty) and *isempty* (a condition to say whether the stack is empty). Letting $x: X$, the axioms are

$top'{=}x\ \wedge\ \neg isempty' \ \Leftarrow\ push\ x$

$ok \ \Leftarrow\ push\ x.\ pop$

$isempty' \ \Leftarrow\ mkempty$

End of Fancy Program-Stack Theory

Once we implement program-stack theory using lists, we know that program-stack theory is consistent if list theory is consistent. Program-stack theory, like data-stack theory, is incomplete. Incompleteness is a freedom for the implementer, who can trade economy against robustness. If we care how this trade will be made, we should strengthen the theory. For example, we could add the axiom

print "error" $\Leftarrow$ *mkempty. pop*

7.1.3 Weak Program-Stack Theory

The program-stack theory we presented first can be weakened and still retain its stack character. We must keep the axiom

$top'{=}x \ \Leftarrow\ push\ x$

but we do not need the composition *push x. pop* to leave all variables unchanged. We do require that any natural number of pushes followed by the same number of pops gives back the original top. The axioms are

$top'{=}top \ \Leftarrow\ balance$

$balance \ \Leftarrow\ ok$

$balance \ \Leftarrow\ push\ x.\ balance.\ pop$

where *balance* is a specification that helps in writing the axioms, but is not an addition to the theory, and does not need to be implemented. To prove an implementation is correct, we must propose a definition for *balance* that uses the implementer's variables, but it doesn't have to be a program. This weaker theory allows an implementation in which popping does not restore the implementer's variable *s* to its pre-pushed value, but instead marks the last item as “garbage”.

A weak theory can be extended in ways that are excluded by a strong theory. For example, we can add the names *count* (of type *nat*) and *start* (a program), with the axioms

$count' = 0 \ \Leftarrow\ start$

$count' = count{+}1 \ \Leftarrow\ push\ x$

$count' = count{+}1 \ \Leftarrow\ pop$

so that *count* counts the number of pushes and pops since the last use of *start*.

End of Weak Program-Stack Theory

7.1.4 Program-Queue Theory

Program-queue theory introduces five names: *mkemptyq* (a program to make the queue empty), *isemptyq* (a condition to say whether the queue is empty), *join* (a procedure with parameter of type X), *leave* (a program), and *front* (of type X). The axioms are

$$isemptyq' \Leftarrow mkemptyq$$
$$isemptyq \Rightarrow front'=x \wedge \neg isemptyq' \Leftarrow join\ x$$
$$\neg isemptyq \Rightarrow front'=front \wedge \neg isemptyq' \Leftarrow join\ x$$
$$isemptyq \Rightarrow (join\ x.\ leave = mkemptyq)$$
$$\neg isemptyq \Rightarrow (join\ x.\ leave = leave.\ join\ x)$$

End of Program-Queue Theory

7.1.5 Program-Tree Theory

As usual, there is more than one way to do it. Imagine a tree that is infinite in all directions; there are no leaves and no root. You are standing at one node in the tree facing one of the three directions *up* (towards the parent of this node), *left* (towards the left child of this node), or *right* (towards the right child of this node). Variable *node* (of type X) tells the value of the item where you are, and it can be assigned a new value. Variable *aim* tells what direction you are facing, and it can be assigned a new direction. Program *go* moves you to the next node in the direction you are facing, and turns you facing back the way you came. For example, we might begin with

$$aim:= up.\ go$$

and then look at *aim* to see where we came from. For later use, we might then assign

$$node:= 3$$

The axioms use an auxiliary specification that helps in writing the axioms, but is not an addition to the theory, and does not need to be implemented: *work* means "Do anything, wander around changing the values of nodes if you like, but do not *go* from this node (your location at the start of *work*) in this direction (the value of variable *aim* at the start of *work*). End where you started, facing the way you were facing at the start.". Here are the axioms.

$$(aim=up) = (aim' \neq up) \Leftarrow go$$
$$node'=node \wedge aim'=aim \Leftarrow go.\ work.\ go$$
$$work \Leftarrow ok$$
$$work \Leftarrow node:= x$$
$$work \Leftarrow a=aim \neq b \wedge (aim:= b.\ go.\ work.\ go.\ aim:= a)$$
$$work \Leftarrow work.\ work$$

Here is another way to define program-trees. Let T (for tree) and p (for pointer) be implementer's variables. The axioms are

$$tree = [tree; X; tree]$$
$$T: tree$$
$$p: *(0, 1, 2)$$
$$node = T@(p; 1)$$
$$change = \langle x: X \rightarrow T:= (p; 1) \rightarrow x \mid T \rangle$$
$$goUp = p:= p_{0;..\leftrightarrow p-1}$$
$$goLeft = p:= p;0$$
$$goRight = p:= p;2$$

If strings and the @ operator are implemented, then this theory is already an implementation. If not, it is still a theory, and should be compared to the previous theory for clarity.

End of Program-Tree Theory

End of Program Theories

7.2 Data Transformation

A program is a specification of computer behavior. Sometimes (but not always) a program is the clearest kind of specification. Sometimes it is the easiest kind of specification to write. If we write a specification as a program, there is no work to implement it. Even though a specification may already be a program, we can, if we like, implement it differently. In some programming languages, implementer's variables are distinguished by being placed inside a "module" or "object", so that changing them is not visible outside the object or module. Perhaps the implementer's variables were chosen to make the specification as clear as possible, but other implementer's variables might be more storage-efficient, or provide faster access on average. Since a theory user has no access to the implementer's variables except through the theory, an implementer is free to change them in any way that provides the same theory to the user. Here's one way.

We can replace the implementer's variables v by new implementer's variables w using a data transformer, which is a boolean expression D relating v and w such that

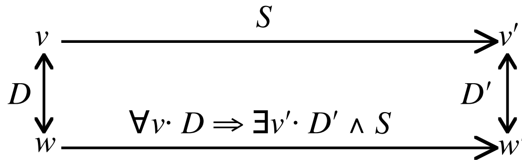
$$\forall w\cdot \exists v\cdot D$$

Here, v and w represent any number of variables. Let D' be the same as D but with primes on all the variables. Then each specification S in the theory is transformed to

$$\forall v\cdot D \Rightarrow \exists v'\cdot D' \land S$$

Specification S talks about its nonlocal variables v (and the user's variables), and the transformed specification talks about its nonlocal variables w (and the user's variables).

Data transformation is invisible to the user. The user imagines that the implementer's variables are initially in state v, and then, according to specification S, they are finally in state v'. Actually, the implementer's variables will initially be in state w related to v by D; the user will be able to suppose they are in a state v because $\forall w\cdot \exists v\cdot D$. The implementer's variables will change state from w to w' according to the transformed specification $\forall v\cdot D \Rightarrow \exists v'\cdot D' \land S$. This says that whatever related initial state v the user was imagining, there is a related final state v' for the user to imagine as the result of S, and so the fiction is maintained. Here is a picture of it.



Implementability of S in its variables (v, v') becomes, via the transformer (D, D'), the new specification in the new variables (w, w').

Our first example is Exercise 363(a). The user's variable is u: *bool* and the implementer's variable is v: *nat*. The theory provides three operations, specified by

$$zero \;=\; v:= 0$$
$$increase \;=\; v:= v+1$$
$$inquire \;=\; u:= even\ v$$

Since the only question asked of the implementer's variable is whether it is even, we decide to replace it by a new implementer's variable w: *bool* according to the data transformer $w = even\ v$. The first operation *zero* becomes

$\forall v\cdot\ w = even\ v \ \Rightarrow\ \exists v'\cdot\ w' = even\ v' \ \land\ (v{:=}\ 0)$

The assignment refers to a state consisting of u and v .

$=\quad \forall v\cdot\ w = even\ v \ \Rightarrow\ \exists v'\cdot\ w' = even\ v' \ \land\ u'{=}u \ \land\ v'{=}0$ One-Point law

$=\quad \forall v\cdot\ w = even\ v \ \Rightarrow\ w' = even\ 0 \ \land\ u'{=}u$ change of variable law, simplify

$=\quad \forall r{:}\ even\ nat\cdot\ w{=}r \ \Rightarrow\ w'{=}\top \ \land\ u'{=}u$ One-Point law

$=\quad w'{=}\top \ \land\ u'{=}u$ The state now consists of u and w .

$=\quad w{:=}\ \top$

Operation *increase* becomes

$\forall v\cdot\ w = even\ v \ \Rightarrow\ \exists v'\cdot\ w' = even\ v' \ \land\ (v{:=}\ v{+}1)$

$=\quad \forall v\cdot\ w = even\ v \ \Rightarrow\ \exists v'\cdot\ w' = even\ v' \ \land\ u'{=}u \ \land\ v'{=}v{+}1$ One-Point law

$=\quad \forall v\cdot\ w = even\ v \ \Rightarrow\ w' = even\ (v{+}1) \ \land\ u'{=}u$ change of variable law, simplify

$=\quad \forall r{:}\ even\ nat\cdot\ w{=}r \ \Rightarrow\ w' = \neg r \ \land\ u'{=}u$ One-Point law

$=\quad w' = \neg w \ \land\ u'{=}u$

$=\quad w{:=}\ \neg w$

Operation *inquire* becomes

$\forall v\cdot\ w = even\ v \ \Rightarrow\ \exists v'\cdot\ w' = even\ v' \ \land\ (u{:=}\ even\ v)$

$=\quad \forall v\cdot\ w = even\ v \ \Rightarrow\ \exists v'\cdot\ w' = even\ v' \ \land\ u' = even\ v \ \land\ v'{=}v$ One-Point law

$=\quad \forall v\cdot\ w = even\ v \ \Rightarrow\ w' = even\ v \ \land\ u' = even\ v$ change of variable law

$=\quad \forall r{:}\ even\ nat\cdot\ w{=}r \ \Rightarrow\ w'{=}r \ \land\ u'{=}r$ One-Point law

$=\quad w'{=}w \ \land\ u'{=}w$

$=\quad u{:=}\ w$

In the previous example, we replaced a bigger state space by a smaller state space. Just to show that it works both ways, here is Exercise 364(a). The user's variable is $u{:}\ bool$ and the implementer's variable is $v{:}\ bool$. The theory provides three operations, specified by

$set \ = \ v{:=}\ \top$

$flip \ = \ v{:=}\ \neg v$

$ask \ = \ u{:=}\ v$

We decide to replace the implementer's variable by a new implementer's variable $w{:}\ nat$ (perhaps for easier access on some computers) according to the data transformer $v = even\ w$. The first operation *set* becomes

$\forall v\cdot\ v = even\ w \ \Rightarrow\ \exists v'\cdot\ v' = even\ w' \ \land\ (v{:=}\ \top)$ One-Point law twice

$=\quad even\ w' \ \land\ u'{=}u$

$\Leftarrow\quad w{:=}\ 0$

Operation *flip* becomes

$\forall v\cdot\ v = even\ w \ \Rightarrow\ \exists v'\cdot\ v' = even\ w' \ \land\ (v{:=}\ \neg v)$ One-Point law twice

$=\quad even\ w' \ne even\ w \ \land\ u'{=}u$

$\Leftarrow\quad w{:=}\ w{+}1$

Operation *ask* becomes

$\forall v\cdot\ v = even\ w \ \Rightarrow\ \exists v'\cdot\ v' = even\ w' \ \land\ (u{:=}\ v)$ One-Point law twice

$=\quad even\ w' = even\ w = u'$

$\Leftarrow\quad u{:=}\ even\ w$

A data transformation does not have to replace all the implementer's variables, and the number of variables being replaced does not have to equal the number of variables replacing them. A data transformation can be done by steps, as a sequence of smaller transformations. A data transformation can be done by parts, as a conjunction of smaller transformations. The next few subsections are examples to illustrate these points.

7.2.0 Security Switch

Exercise 367 is to design a security switch. It has three boolean user's variables a , b , and c . The users assign values to a and b as input to the switch. The switch's output is assigned to c . The output changes when both inputs have changed. More precisely, the output changes when both inputs differ from what they were the previous time the output changed. The idea is that one user might flip their input indicating a desire for the output to change, but the output does not change until the other user flips their input indicating agreement that the output should change. If the first user changes back before the second user changes, the output does not change.

We can implement the switch with two boolean implementer's variables:

A records the state of input a at last output change
B records the state of input b at last output change

There are two operations:

$a:= \neg a.\ \ \mathbf{if}\ a \ne A \land b \ne B\ \mathbf{then}\ (c:= \neg c.\ \ A:= a.\ \ B:= b)\ \mathbf{else}\ ok$
$b:= \neg b.\ \ \mathbf{if}\ a \ne A \land b \ne B\ \mathbf{then}\ (c:= \neg c.\ \ A:= a.\ \ B:= b)\ \mathbf{else}\ ok$

In each operation, a user flips their input variable, and the switch checks if this input assignment makes both inputs differ from what they were at last output change; if so, the output is changed, and the current input values are recorded. This implementation is a direct formalization of the problem, but it can be simplified by data transformation.

We replace implementer's variables A and B by nothing according to the transformer

$A=B=c$

To check that this is a transformer, we check

$\exists A, B\cdot\ A=B=c$ — generalization, using c for both A and B
$\Leftarrow\ \ \top$

There are no new variables, so there was no universal quantification. The transformation does not affect the assignments to a and b , so we have only one transformation to make.

$\forall A, B\cdot\ \ A=B=c$
$\quad \Rightarrow \exists A', B'\cdot\ \ A'=B'=c'$
$\qquad \land\ \mathbf{if}\ a \ne A \land b \ne B\ \mathbf{then}\ (c:= \neg c.\ \ A:= a.\ \ B:= b)\ \mathbf{else}\ ok$

expand assignments and ok

$=\ \ \forall A, B\cdot\ \ A=B=c$
$\quad \Rightarrow \exists A', B'\cdot\ \ A'=B'=c'$
$\qquad \land\ \mathbf{if}\ a \ne A \land b \ne B\ \mathbf{then}\ (a'=a \land b'=b \land c'=\neg c \land A'=a \land B'=b)$
$\qquad\quad \mathbf{else}\ (a'=a \land b'=b \land c'=c \land A'=A \land B'=B)$

one-point for A' and B'

$=\ \ \forall A, B\cdot\ A=B=c\ \Rightarrow\ \ \mathbf{if}\ a \ne A \land b \ne B\ \mathbf{then}\ (a'=a \land b'=b \land c'=\neg c \land c'=a \land c'=b)$
$\qquad \mathbf{else}\ (a'=a \land b'=b \land c'=c \land c'=A \land c'=B)$

one-point for A and B

$=\ \ \mathbf{if}\ a \ne c \land b \ne c\ \mathbf{then}\ (a'=a \land b'=b \land c'=\neg c \land c'=a \land c'=b)$
$\quad \mathbf{else}\ (a'=a \land b'=b \land c'=c \land c'=c \land c'=c)$

use **if**-part as context to change **then**-part

$=\ \ \mathbf{if}\ a \ne c \land b \ne c\ \mathbf{then}\ (a'=a \land b'=b \land c'=\neg c \land c'=\neg c \land c'=\neg c)$
$\quad \mathbf{else}\ (a'=a \land b'=b \land c'=c \land c'=c \land c'=c)$
$=\ \ \mathbf{if}\ a \ne c \land b \ne c\ \mathbf{then}\ c:= \neg c\ \mathbf{else}\ ok$
$=\ \ c:= (a \ne c \land b \ne c) \ne c$

Output c becomes the majority value of a , b , and c . (As a circuit, that's three “exclusive or” gates and one “and” gate.)

End of Security Switch

7.2.1 Take a Number

The next example is Exercise 370 (take a number): Maintain a list of natural numbers standing for those that are "in use". The three operations are:

- make the list empty (for initialization)
- assign to variable n a number that is not in use, and add this number to the list (now it is in use)
- given a number n that is in use, remove it from the list (now it is no longer in use, and it can be reused later)

The user's variable is n: nat . Although the exercise talks about a list, we see from the operations that the items are always distinct, their order is irrelevant, and there is no nesting structure; that suggests using a bunch variable. But we will need to quantify over this variable, so we need it to be an element. We therefore use a set variable $s \subseteq \{nat\}$ as our implementer's variable. The three operations are

$$start \;=\; s'=\{null\}$$
$$take \;=\; \neg\, n'{\in}s \;\land\; s' = s{\cup}\{n'\}$$
$$give \;=\; n{\in}s \;\Rightarrow\; \neg\, n{\in}s' \;\land\; s'{\cup}\{n\} = s$$

Here is a data transformation that replaces set s with natural m according to the transformer

$$s \subseteq \{0,..m\}$$

Instead of maintaining the exact set of numbers that are in use, we will maintain a possibly larger set. We will still never give out a number that is in use. We transform $start$ as follows.

$$\forall s\cdot\ s{\subseteq}\{0,..m\} \;\Rightarrow\; \exists s'\cdot\ s'{\subseteq}\{0,..m'\} \land s'=\{null\} \qquad \text{one-point and identity}$$
$$=\quad \top$$
$$\Leftarrow\quad ok$$

The transformed specification is just $\top$, which is most efficiently refined as ok . Since s is only a subset of $\{0,..m\}$, not necessarily equal to $\{0,..m\}$, it does not matter what m is; we may as well leave it alone. Operation $take$ is transformed as follows.

$$\forall s\cdot\ s{\subseteq}\{0,..m\} \;\Rightarrow\; \exists s'\cdot\ s'{\subseteq}\{0,..m'\} \;\land\; \neg\, n'{\in}s \;\land\; s' = s{\cup}\{n'\}$$

several omitted steps

$$=\quad m \le n' < m'$$
$$\Leftarrow\quad n{:=}\ m.\ \ m{:=}\ m{+}1$$

Operation $give$ is transformed as follows.

$$\forall s\cdot\ s{\subseteq}\{0,..m\} \;\Rightarrow\; \exists s'\cdot\ s'{\subseteq}\{0,..m'\} \;\land\; (n{\in}s \;\Rightarrow\; \neg\, n{\in}s' \;\land\; s'{\cup}\{n\} = s)$$

several omitted steps

$$=\quad (n{+}1 = m \Rightarrow n \le m') \land (n{+}1 < m \Rightarrow m \le m')$$
$$\Leftarrow\quad ok$$

Thanks to the data transformation, we have an extremely efficient solution to the problem. One might argue that we have not solved the problem at all, because we do not maintain a list of numbers that are "in use". But who can tell? The only use made of the list is to obtain a number that is not currently in use, and that service is provided.

Our implementation of the "take a number" problem corresponds to the "take a number" machines that are common at busy service centers. Now suppose we want to provide two "take a number" machines that can operate independently. We might try replacing s with two variables i, j: nat according to the transformer $s \subseteq \{0,..max\ i\ j\}$. Operation $take$ becomes

$$\forall s\cdot\ s{\subseteq}\{0,..max\ i\ j\} \;\Rightarrow\; \exists s'\cdot\ s'{\subseteq}\{0,..max\ i'\ j'\} \;\land\; \neg\, n'{\in}s \;\land\; s' = s{\cup}\{n'\}$$

several omitted steps

$$=\quad max\ i\ j \le n' < max\ i'\ j'$$
$$\Leftarrow\quad n{:=}\ max\ i\ j.\ \ \textbf{if}\ i{\ge}j\ \textbf{then}\ i{:=}\ i{+}1\ \textbf{else}\ j{:=}\ j{+}1$$

From the program on the last line we see that this data transformation does not provide the independent operation of two machines as we were hoping. Perhaps a different data transformation will work better. Let's put the even numbers on one machine and the odd numbers on the other. The new variables are i: $2\times nat$ and j: $2\times nat+1$. The transformer is

$$\forall k{:}\ \sim s\cdot\ even\ k \wedge k<i\ \vee\ odd\ k \wedge k<j$$

Now *take* becomes

$$\forall s\cdot\quad (\forall k{:}\ \sim s\cdot\ even\ k \wedge k<i\ \vee\ odd\ k \wedge k<j)$$
$$\Rightarrow\ \exists s'\cdot\ (\forall k{:}\ \sim s'\cdot\ even\ k \wedge k<i'\ \vee\ odd\ k \wedge k<j')\ \wedge\ \neg\ n'{\in}s\ \wedge\ s' = s\cup\{n'\}$$

several omitted steps

$$=\quad even\ n'\ \wedge\ i \le n' < i'\ \ \vee\ \ odd\ n'\ \wedge\ j \le n' < j'$$
$$\Leftarrow\quad (n{:=}\ i.\ \ i{:=}\ i+2)\ \vee\ (n{:=}\ j.\ \ j{:=}\ j+2)$$

Now we have a "distributed" solution to the problem: we can take a number from either machine without disturbing the other. The price of the distribution is that we have lost all fairness between the two machines; a recently arrived customer using one machine may be served before an earlier customer using the other machine.

End of Take a Number

7.2.2 Parsing

Exercise 362 (parsing): Define E as a bunch of strings of lists of characters satisfying

$$E\ =\ ["x"],\ \ ["if"]; E; ["then"]; E; ["else"]; E$$

Given a string of lists of characters, write a program to determine if the string is in the bunch E .

For the problem to be nontrivial, we assume that recursive data definition and bunch inclusion are not implemented. The solution will have to be a search, so we need a variable to represent the bunch of strings still in contention, beginning with all the strings in E , eliminating strings as we go, and ending either when the given string is found or when none of the remaining strings is the given string.

Let the given string be s (a constant). Our first decision is to parse from left to right, so we introduce natural variable n , increasing from 0 to at most $\leftrightarrow s$, indicating how much of s we have parsed. Let A be a variable whose value is a bunch of strings of lists of characters. Bunch A will consist of all strings in E that might possibly be s according to what we have seen of s . We can express the result as the final value of boolean variable q .

To reduce the number of cases that we have to consider, we will use two sentinels. We assume that s ends with the sentinel ["eos"] (end of string); this is an item that cannot appear anywhere except at the end of s (some programming languages provide this sentinel automatically). And when we initialize variable A , we will add the sentinel ["eog"] (end of grammar) to the end of every string, and assume that ["eog"] cannot appear anywhere except at the end of strings in A . The problem and its refinement are as follows:

$$q' = (s_{0;..\leftrightarrow s-1} : E)\ \ \Leftarrow\ \ A{:=}\ E;["eog"].\ \ n{:=}\ 0.\ \ P$$

where $P\ =\ n{\le}{\leftrightarrow}s\ \wedge\ A_{0;..n} = s_{0;..n}\ \Rightarrow\ q' = (s_{0;..\leftrightarrow s-1};["eog"] : A)$. In words, the new problem P says that if the strings in A look like s up to index n , then the question is whether s is in A (with a suitable adjustment of sentinels). The proof of this refinement uses the fact that E is a nonempty bunch, but we will not need the fact that E is a bunch of nonempty strings. Here is the refinement of the remaining problem.

$P \Leftarrow$ **if** s_n: A_n **then** $(A{:=}\ (\S a{:}\ A\cdot\ a_n = s_n).\ \ n{:=}\ n{+}1.\ \ P)$
else $q{:=}$ ["eog"]: $A_n\ \wedge\ s_n$=["eos"]

From P we know that all strings in A are identical to s up to index n . If there are strings in A that agree with s at index n , then we reduce bunch A to just those strings, and move along one index. If not, then either we have run out of candidates and we should assign $\perp$ to q , or we have come to the end of s and also to the end of one of the candidates and we should assign $\top$ to q . We omit the proofs of these refinements in order to pursue our current topic, data transformation.

We now replace variable A with variable b whose value is a single string of lists of characters. We represent bunch E with ["E"] , which we assume cannot be in the given string s . (In parsing theory "E" is called a “nonterminal”.) For example, the string

["if"]; ["x"]; ["then"]; ["E"]; ["else"]; ["E"]

represents the bunch of strings

["if"]; ["x"]; ["then"]; E; ["else"]; E

The data transformer is, informally,

A = (b with all occurrences of item ["E"] replaced by bunch E)

Let Q be the result of transforming P . The result of the transformation is as follows.

$q' = (s_{0;..\leftrightarrow s-1} : E)\ \Leftarrow\ b{:=}$ ["E"];["eog"]. $n{:=}\ 0.\ \ Q$

$Q \Leftarrow$ **if** $s_n{=}b_n$ **then** $(n{:=}\ n{+}1.\ \ Q)$
else if b_n=["E"] $\wedge$ s_n=["x"] **then** ($b{:=}\ b_{0;..n}$;["x"];$b_{n+1;..\leftrightarrow b}$. $n{:=}\ n{+}1.\ \ Q$)
else if b_n=["E"] $\wedge$ s_n=["if"]
then ($b{:=}\ b_{0;..n}$;["if"];["E"];["then"];["E"];["else"];["E"];$b_{n+1;..\leftrightarrow b}$. $n{:=}\ n{+}1.\ \ Q$)
else $q{:=}$ b_n=["eog"] $\wedge$ s_n=["eos"]

We can make a minor improvement by changing the representation of E from ["E"] to ["x"] ; then one of the cases disappears, and we get

$q' = (s_{0;..\leftrightarrow s-1} : E)\ \Leftarrow\ b{:=}$ ["x"];["eog"]. $n{:=}\ 0.\ \ Q$

$Q \Leftarrow$ **if** $s_n{=}b_n$ **then** $(n{:=}\ n{+}1.\ \ Q)$
else if b_n=["x"] $\wedge$ s_n=["if"]
then ($b{:=}\ b_{0;..n}$;["if"];["x"];["then"];["x"];["else"];["x"];$b_{n+1;..\leftrightarrow b}$. $n{:=}\ n{+}1.\ \ Q$)
else $q{:=}$ b_n=["eog"] $\wedge$ s_n=["eos"]

Our next improvement is to notice that we don't need the initial portion of b , which is identical to the initial portion of s . So we transform again, replacing b with c using the transformer

$b\ =\ s_{0;..n};c$

Let R be the result of transforming Q . The result of the transformation is as follows.

$q' = (s_{0;..\leftrightarrow s-1} : E)\ \Leftarrow\ c{:=}$ ["x"];["eog"]. $n{:=}\ 0.\ \ R$

$R \Leftarrow$ **if** $s_n{=}c_0$ **then** $(c{:=}\ c_{1;..\leftrightarrow c}.\ \ n{:=}\ n{+}1.\ \ R)$
else if c_0=["x"] $\wedge$ s_n=["if"] **then** ($c{:=}$ ["x"];["then"];["x"];["else"];c. $n{:=}\ n{+}1.\ \ R$)
else $q{:=}$ c_0=["eog"] $\wedge$ s_n=["eos"]

Variable c behaves as a stack, so we could replace it by stack operations.

End of Parsing

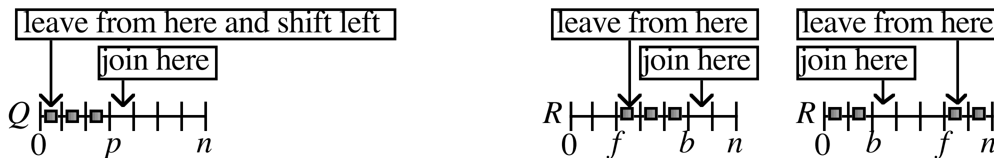
7.2.3 Limited Queue

The next example, Exercise 371, transforms a limited queue to achieve a time bound that is not met by the original implementation. A limited queue is a queue with a limited number of places for items. Let the limit be positive natural n , and let Q: $[n*X]$ and p: *nat* be implementer's variables. Then the original implementation is as follows.

$$mkemptyq = p:= 0$$
$$isemptyq = p=0$$
$$isfullq = p=n$$
$$join\ x = Qp:= x.\ p:= p+1$$
$$leave = \textbf{for}\ i:= 1;..p\ \textbf{do}\ Q(i-1):= Qi.\ p:= p-1$$
$$front = Q0$$

A user of this theory would be well advised to precede any use of *join* with the test $\neg isfullq$, and any use of *leave* or *front* with the test $\neg isemptyq$.

A new item joins the back of the queue at position p taking zero time (measured recursively) to do so. The front item is always found instantly at position 0 . Unfortunately, removing the front item from the queue takes time $p-1$ to shift all remaining items down one index. We want to transform the queue so that all operations are instant. Variables Q and p will be replaced by R: $[n*X]$ and f, b: $0,..n$ with f and b indicating the current front and back.



The idea is that b and f move cyclically around the list; when f is to the left of b the queue items are between them; when b is to the left of f the queue items are in the outside portions. Here is the data transformer D .

$$0 \le p = b-f < n \ \wedge\ Q[0;..p] = R[f;..b]$$
$$\vee\ \ 0 < p = n-f+b \le n \ \wedge\ Q[0;..p] = R[(f;..n);\ (0;..b)]$$

Now we transform. First *mkemptyq* .

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge p'=0 \wedge Q'=Q \qquad \text{several omitted steps}$$
$$= \quad f'=b'$$
$$\Leftarrow \quad f:= 0.\ b:= 0$$

Next we transform *isemptyq* . Although *isemptyq* happens to be boolean and can be interpreted as an unimplementable specification, its purpose (like *front* , which isn't boolean) is to tell the user about the state of the queue. We don't transform arbitrary expressions; we transform implementable specifications (usually programs). So we suppose c is a user's variable, and transform $c:= isemptyq$.

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \wedge c'=(p=0) \wedge p'=p \wedge Q'=Q \qquad \text{several omitted steps}$$
$$= \quad f<b \ \wedge\ f'<b' \ \wedge\ b-f = b'-f' \ \wedge\ R[f;..b] = R'[f';..b'] \ \wedge\ \neg c'$$
$$\vee \quad f<b \ \wedge\ f'>b' \ \wedge\ b-f = n+b'-f' \ \wedge\ R[f;..b] = R'[(f';..n);\ (0;..b')] \ \wedge\ \neg c'$$
$$\vee \quad f>b \ \wedge\ f'<b' \ \wedge\ n+b-f = b'-f' \ \wedge\ R[(f;..n);\ (0;..b)] = R'[f';..b'] \ \wedge\ \neg c'$$
$$\vee \quad f>b \ \wedge\ f'>b' \ \wedge\ b-f = b'-f' \ \wedge\ R[(f;..n);\ (0;..b)]=R'[(f';..n);\ (0;..b')] \ \wedge\ \neg c'$$

Initially R might be in the "inside" or "outside" configuration, and finally R' might be either way, so that gives us four disjuncts. Very suspiciously, we have $\neg c'$ in every case. That's because $f=b$ is missing! So the transformed operation is unimplementable. That's the transformer's way of telling us that the new variables do not hold enough information to answer whether the queue is empty. The problem occurs when $f=b$ because that could be either an empty queue or a full queue. A solution is to add a new variable m: *bool* to say whether we have the "inside" mode or "outside" mode. We revise the transformer D as follows:

$$m \land 0 \le p = b-f < n \land Q[0;..p] = R[f;..b]$$
$$\lor \quad \neg m \land 0 < p = n-f+b \le n \land Q[0;..p] = R[(f;..n); (0;..b)]$$

Now we have to retransform *mkemptyq* .

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land p'=0 \land Q'=Q \qquad \text{several omitted steps}$$
$$= \quad m' \land f'=b'$$
$$\Leftarrow \quad m:= \top.\ f:= 0.\ b:= 0$$

Next we transform $c:=$ *isemptyq* .

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land c'=(p=0) \land p'=p \land Q'=Q \qquad \text{several omitted steps}$$
$$= \quad m \land f<b \land m' \land f'<b' \land b-f = b'-f' \land R[f;..b] = R'[f';..b'] \land \neg c'$$
$$\lor \quad m \land f<b \land \neg m' \land f'>b' \land b-f = n+b'-f'$$
$$\land R[f;..b] = R'[(f';..n); (0;..b')] \land \neg c'$$
$$\lor \quad \neg m \land f>b \land m' \land f'<b' \land n+b-f = b'-f'$$
$$\land R[(f;..n); (0;..b)] = R'[f';..b'] \land \neg c'$$
$$\lor \quad \neg m \land f>b \land \neg m' \land f'>b' \land b-f = b'-f'$$
$$\land R[(f;..n); (0;..b)] = R'[(f';..n); (0;..b')] \land \neg c'$$
$$\lor \quad m \land f=b \land m' \land f'=b' \land c'$$
$$\lor \quad \neg m \land f=b \land \neg m' \land f'=b' \land R[(f;..n); (0;..b)]=R'[(f';..n); (0;..b')] \land \neg c'$$
$$\Leftarrow \quad c' = (m \land f=b) \land f'=f \land b'=b \land R'=R$$
$$= \quad c:= m \land f=b$$

The transformed operation offered us the opportunity to rotate the queue within R , but we declined to do so. For other data structures, it is sometimes a good strategy to reorganize the data structure during an operation, and data transformation always tells us what reorganizations are possible. Each of the remaining transformations offers the same opportunity, but there is no reason to rotate the queue, and we decline each time.

Next we transform $c:=$ *isfullq* , *join* x , and *leave* .

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land c'=(p=n) \land p'=p \land Q'=Q \qquad \text{several omitted steps}$$
$$\Leftarrow \quad c:= \neg m \land f=b$$

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land Q'=Q[0;..p]+[x]+Q[p+1;..n] \land p'=p+1$$
several omitted steps
$$\Leftarrow \quad Rb:= x.\ \textbf{if } b+1=n \textbf{ then } (b:= 0.\ m:= \bot) \textbf{ else } b:= b+1$$

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land Q'=Q[(1;..p); (p-1;..n)] \land p'=p-1 \qquad \text{several omitted steps}$$
$$\Leftarrow \quad \textbf{if } f+1=n \textbf{ then } (f:= 0.\ m:= \top) \textbf{ else } f:= f+1$$

Last we transform $x:=$ *front* where x is a user's variable of the same type as the items.

$$\forall Q, p\cdot D \Rightarrow \exists Q', p'\cdot D' \land x'=Q0 \land p'=p \land Q'=Q \qquad \text{several omitted steps}$$
$$\Leftarrow \quad x:= Rf$$

End of Limited Queue

7.2.4 Soundness and Completeness optional

Data transformation is sound in the sense that a user cannot tell that a transformation has been made; that was the criterion of its design. But it is possible to find two programs that behave identically from a user's view, but for which there is no data transformer to transform one into the other. In that sense, data transformation is incomplete.

Exercise 374 illustrates the problem. The user's variable is i and the implementer's variable is j, both of type nat. The operations are:

$$initialize \;=\; i' = 0 \le j' < 3$$

$$step \;=\; \textbf{if}\ j>0\ \textbf{then}\ (i:= i+1.\ \ j:= j-1)\ \textbf{else}\ ok$$

The user can look at i but not at j. The user can $initialize$, which starts i at 0 and starts j at any of 3 values. The user can then repeatedly $step$ and observe that i increases 0 or 1 or 2 times and then stops increasing, which effectively tells the user what value j started with.

If this were a practical problem, we would notice that $initialize$ can be refined, resolving the nondeterminism. For example,

$$initialize \;\Leftarrow\; i:= 0.\ \ j:= 0$$

We could then transform $initialize$ and $step$ to get rid of j, replacing it with nothing. The transformer is $j=0$. It transforms the implementation of $initialize$ as follows:

$$\forall j\cdot\ j=0 \Rightarrow \exists j'\cdot\ j'=0 \land i'=j'=0$$

$$=\quad i:= 0$$

And it transforms $step$ as follows:

$$\forall j\cdot\ j=0 \Rightarrow \exists j'\cdot\ j'=0 \ \land\ \textbf{if}\ j>0\ \textbf{then}\ (i:= i+1.\ \ j:= j-1)\ \textbf{else}\ ok$$

$$=\quad ok$$

If this were a practical problem, we would be done. But the theoretical problem is to replace j with boolean variable b without resolving the nondeterminism, so that

$initialize$ is transformed to $i'=0$

$step$ is transformed to $\textbf{if}\ b \land i<2\ \textbf{then}\ i' = i+1\ \textbf{else}\ ok$

Now the transformed $initialize$ starts b either at $\top$, meaning that i will be increased, or at $\bot$, meaning that i will not be increased. Each use of the transformed $step$ tests b to see if we might increase i, and checks $i<2$ to ensure that i will remain below 3. If i is increased, b is again assigned either of its two values. The user will see i start at 0 and increase 0 or 1 or 2 times and then stop increasing, exactly as in the original specification. The nondeterminism is maintained. But there is no transformer in variables i, j, and b to do the job.

End of Soundness and Completeness

End of Data Transformation

End of Theory Design and Implementation

8 Concurrency

Concurrency, also known as parallelism, means two or more activities occurring at the same time. In some other books, the words "concurrency" and "parallelism" are used to mean that the activities occur in an unspecified sequence, or that they are composed of smaller activities that occur in an interleaved sequence. But in this book they mean that there is more than one activity at a time.

8.0 Independent Composition

We define the independent composition of specifications P and Q so that $P||Q$ (pronounced " P parallel Q ") is satisfied by a computer that behaves according to P and, at the same time, in parallel, according to Q. The operands of $||$ are called processes.

When we defined the dependent composition of P and Q, we required that P and Q have exactly the same state variables, so that we could identify the final state of P with the initial state of Q. For independent composition $P||Q$, we require that P and Q have completely different state variables, and the state variables of the composition $P||Q$ are those of both P and Q. If we ignore time and space, independent composition is conjunction.

$$P||Q \;=\; P \land Q$$

When we decide to create an independent composition, we decide how to partition the variables. Given specification S, if we choose to refine it as $S \Leftarrow P||Q$, we have to decide which variables of S belong to P, and which to Q. For example, in variables x, y, and z, the specification

$$x' = x+1 \;\land\; y' = y+2 \;\land\; z'=z$$

can be refined by the independent composition

$$x:= x+1 \;||\; y:= y+2$$

if we partition the variables. Clearly x has to belong to the left process for the assignment to x to make sense, and similarly y has to belong to the right process. As for z, it doesn't matter which process we give it to; either way

$$x:= x+1 \;||\; y:= y+2 \;=\; x' = x+1 \;\land\; y' = y+2 \;\land\; z'=z$$

If we are presented with an independent composition, and we are not told how the variables are partitioned, we have to determine a partitioning that makes sense. Here's a way that usually works: If either x' or $x:=$ appears in a process specification, then x belongs to that process. If neither x' nor $x:=$ appears at all, then x can be placed on either side of the partition. This way of partitioning does not work when x' or $x:=$ appears in both process specifications.

In the next example

$$x:= y \;||\; y:= x$$

again x belongs to the left process, y to the right process, and z to either process. In the left process, y appears, but neither y' nor $y:=$ appears, so y is a state constant, not a state variable, in the left process. Similarly x is a state constant in the right process. And the result is

$$x:= y \;||\; y:= x \;=\; x'=y \land y'=x \land z'=z$$

Variables x and y swap values, apparently without a temporary variable. In fact, an implementation of a process will have to make a private copy of the initial value of a variable belonging to the other process if the other process contains an assignment to that variable.

In boolean variable b and integer variable x ,

$$b{:=}\ x{=}x \ \| \ x{:=}\ x{+}1 \qquad \text{replace } x{=}x \text{ by } \top$$
$$= \quad b{:=}\ \top \ \| \ x{:=}\ x{+}1$$

On the first line, it may seem possible for the process on the right side to increase x between the two evaluations of x in the left process, resulting in the assignment of $\bot$ to b . And that would be a mathematical disaster; we could not even be sure $x{=}x$. According to the last line, this does not happen; both occurrences of x in the left process refer to the initial value of variable x . We can use the reflexive and transparent axioms of equality, and replace $x{=}x$ by $\top$.

In a dependent composition as defined in Chapter 4, the intermediate values of variables are local to the dependent composition; they are hidden by the quantifier $\exists x'', y'', \cdots$. If one process is a dependent composition, the other cannot see its intermediate values. For example,

$$(x{:=}\ x{+}1.\ x{:=}\ x{-}1) \ \| \ y{:=}\ x$$
$$= \quad ok \,\|\, y{:=}\ x$$
$$= \quad y{:=}\ x$$

On the first line, it may seem possible for the process on the right side to evaluate x between the two assignments to x in the left process. According to the last line, this does not happen; the occurrence of x in the right process refers to the initial value of variable x . In the next chapter we introduce interactive variables and communication channels between processes so they can see the intermediate values of each other's variables, but in this chapter processes are not able to interact.

In the previous example, we replaced $(x{:=}\ x{+}1.\ x{:=}\ x{-}1)$ by ok . And of course we can make the reverse replacement whenever x is one of the state variables. Although x is one of the variables of the composition

$$ok \,\|\, x{:=}\ 3$$

it is not one of the variables of the left process ok due to the assignment in the right process. So we cannot equate that composition to

$$(x{:=}\ x{+}1.\ x{:=}\ x{-}1) \ \| \ x{:=}\ 3$$

Sometimes the need for shared memory arises from poor program structure. For example, suppose we decide to have two processes, as follows.

$$(x{:=}\ x{+}y.\ x{:=}\ x{\times}y)$$
$$\| \quad (y{:=}\ x{-}y.\ y{:=}\ x/y)$$

The first modifies x twice, and the second modifies y twice. But suppose we want the second assignment in each process to use the values of x and y after the first assignments of both processes. This may seem to require not only a shared memory, but also synchronization of the two processes at their mid-points, forcing the faster process to wait for the slower one, and then to allow the two processes to continue with the new, updated values of x and y . Actually, it requires neither shared memory nor synchronization devices. It is achieved by writing

$$(x{:=}\ x{+}y \ \| \ y{:=}\ x{-}y).\ (x{:=}\ x{\times}y \ \| \ y{:=}\ x/y)$$

So far, independent composition is just conjunction, and there is no need to introduce a second symbol $\|$ for conjunction. But now we consider time. The time variable is not subject to partitioning; it belongs to both processes. In $P\|Q$, both P and Q begin execution at time t , but their executions may finish at different times. Execution of the composition $P\|Q$ finishes when both P and Q are finished. With time, independent composition is defined as

$$P\|Q \ = \ \exists t_P, t_Q\cdot\ \langle t'\to P\rangle t_P \ \wedge \ \langle t'\to Q\rangle t_Q \ \wedge \ t' = max\ t_P\ t_Q$$
$$= \ \exists t_P, t_Q\cdot \quad (\text{substitute } t_P \text{ for } t' \text{ in } P)$$
$$\wedge \ (\text{substitute } t_Q \text{ for } t' \text{ in } Q)$$
$$\wedge \ t' = max\ t_P\ t_Q$$

8.0.0 Laws of Independent Composition

Let x and y be different state variables, let e , f , and b be expressions of the prestate, and let P , Q , R , and S be specifications. Then

$(x:= e \| y:=f).\ P \;=\;$ (for x substitute e and independently for y substitute f in P) — independent substitution

$P \| Q \;=\; Q \| P$ — symmetry

$P \| (Q \| R) \;=\; (P \| Q) \| R$ — associativity

$P \| ok \;=\; ok \| P \;=\; P$ — identity

$P \| Q \vee R \;=\; (P \| Q) \vee (P \| R)$ — distributivity

$P \| \textbf{if } b \textbf{ then } Q \textbf{ else } R \;=\; \textbf{if } b \textbf{ then } (P \| Q) \textbf{ else } (P \| R)$ — distributivity

$\textbf{if } b \textbf{ then } (P\|Q) \textbf{ else } (R\|S) \;=\; \textbf{if } b \textbf{ then } P \textbf{ else } R \| \textbf{if } b \textbf{ then } Q \textbf{ else } S$ — distributivity

The Associative Law says we can compose any number of processes without worrying how they are grouped. As an example of the Substitution Law,

$$(x:= x+y \| y:= x\times y).\ z' = x-y \;=\; z' = (x+y) - (x\times y)$$

Note that each substitution replaces all and only the original occurrences of its variable. This law generalizes the earlier Substitution Law from one variable to two, and it can be generalized further to any number of variables.

Refinement by Steps works for independent composition:

If $A \Leftarrow B\|C$ and $B \Leftarrow D$ and $C \Leftarrow E$ are theorems, then $A \Leftarrow D\|E$ is a theorem.

So does Refinement by Parts:

If $A \Leftarrow B\|C$ and $D \Leftarrow E\|F$ are theorems, then $A\wedge D \Leftarrow B\wedge E \| C\wedge F$ is a theorem.

End of Laws of Independent Composition

8.0.1 List Concurrency

We have defined independent composition by partitioning the variables. For finer-grained concurrency, we can extend this same idea to the individual items within list variables. In Chapter 5 we defined assignment to a list item as

$$Li:= e \;=\; L'i=e \;\wedge\; (\forall j\cdot j\neq i \Rightarrow L'j=Lj) \;\wedge\; x'=x \;\wedge\; y'=y \;\wedge\; ...$$

which says not only that the assigned item has the right final value, but also that all other items and all other variables do not change value. For independent composition, we must specify the final values of only the items and variables in one side of the partition.

As a good example of list concurrency, we do Exercise 140: find the maximum item in a list. The maximum of a list is easily expressed with the MAX quantifier, but we will assume MAX is not implemented. The easiest and simplest solution is probably functional, with parallelism coming from the fact that the arguments of a function (operands of an operator) can always be evaluated in parallel. To use our parallel operator, we present an imperative solution. Let L be the list whose maximum item is sought. If L is an empty list, its maximum is $-\infty$; assume that L is nonempty. Assume further that L is a variable whose value is not wanted after we know its maximum (we'll remove this assumption later). Our specification will be $L'\ 0 = MAX\ L$; at the end, item 0 of list L will be the maximum of all original items. The first step is to generalize from the maximum of a nonempty list to the maximum of a nonempty segment of a list. So define

$$findmax \;=\; \langle i, j\rightarrow i<j \Rightarrow L'\ i = MAX\ L\ [i;..j]\rangle$$

Our specification is $findmax\ 0\ (\#L)$. We refine as follows.

$$findmax\ i\ j \Leftarrow \ \textbf{if}\ j{-}i = 1\ \textbf{then}\ ok\ \textbf{else}\ (\ (findmax\ i\ (div\ (i{+}j)\ 2)\ \|\ findmax\ (div\ (i{+}j)\ 2)\ j).\ L\ i := max\ (L\ i)\ (L\ (div\ (i{+}j)\ 2))\ )$$

If $j{-}i = 1$ the segment contains one item; to place the maximum item (the only item) at index i requires no change. In the other case, the segment contains more than one item; we divide the segment into two halves, placing the maximum of each half at the beginning of the half. In the parallel composition, the two processes $findmax\ i\ (div\ (i{+}j)\ 2)$ and $findmax\ (div\ (i{+}j)\ 2)\ j$ change disjoint segments of the list. We finish by placing the maximum of the two maximums at the start of the whole segment. The recursive execution time is $ceil\ (log\ (j{-}i))$, exactly the same as for binary search, which this program closely resembles.

If list L must remain constant, we can use a new list M of the same type as L to collect our partial results. We redefine

$$findmax = \langle i, j \rightarrow i{<}j \Rightarrow M'\ i = MAX\ L\ [i;..j] \rangle$$

and in the program we change ok to $M\ i := L\ i$ and we change the final assignment to

$$M\ i := max\ (M\ i)\ (M\ (div\ (i{+}j)\ 2))$$

End of List Concurrency

End of Independent Composition

8.1 Sequential to Parallel Transformation

The goal of this section is to transform programs without concurrency into programs with concurrency. A simple example illustrates the idea. Ignoring time,

$$x := y.\ x := x{+}1.\ z := y$$
$$= \quad x := y.\ (x := x{+}1\ \|\ z := y)$$
$$= \quad (x := y.\ x := x{+}1)\ \|\ z := y$$

Execution of the program on the first line can be depicted as follows.

start ⟶ $x := y$ ⟶ $x := x{+}1$ ⟶ $z := y$ ⟶ finish

The first two assignments cannot be executed concurrently, but the last two can, so we transform the program. Execution can now be depicted as

start ⟶ $x := y$ ⟶ ($x := x{+}1$ and $z := y$) ⟶ finish

Now we have the first and last assignments next to each other, in sequence; they too can be executed concurrently. Execution can be

start ⟶ ($x := y$ ⟶ $x := x{+}1$ and $z := y$) ⟶ finish

Whenever two programs occur in sequence, and neither assigns to any variable assigned in the other, and no variable assigned in the first appears in the second, they can be placed in parallel; a copy must be made of the initial value of any variable appearing in the first and assigned in the second. Whenever two programs occur in sequence, and neither assigns to any variable appearing

in the other, they can be placed in parallel without any copying of initial values. This transformation does not change the result of a computation, but it may decrease the time, and that is the reason for doing it.

Program transformation to obtain concurrency can often be performed automatically by the implementation. Sometimes it can only be performed by the implementation because the result is not expressible as a source program.

8.1.0 Buffer

Consider two programs, *produce* and *consume*, whose only common variable is b. *produce* assigns to b and *consume* uses the value of b.

$$produce \;=\; \cdots\cdots b{:=}\, e\cdots\cdots$$
$$consume \;=\; \cdots\cdots x{:=}\, b\cdots\cdots$$

These two programs are executed alternately, repeatedly, forever.

$$control \;=\; produce.\ consume.\ control$$

Using P for *produce* and C for *consume*, execution looks like this:

$$P \longrightarrow C \longrightarrow P \longrightarrow C \longrightarrow P \longrightarrow C \longrightarrow P \longrightarrow C \longrightarrow$$

Many programs have producer and consumer components somewhere in them. Variable b is called a buffer; it may be a large data structure. The idea is that *produce* and *consume* are time-consuming, and we can save time if we put them in parallel. As they are, we cannot put them in parallel because the first assigns to b and the second uses b. So we unroll the loop once.

$$control \;=\; produce.\ newcontrol$$
$$newcontrol \;=\; consume.\ produce.\ newcontrol$$

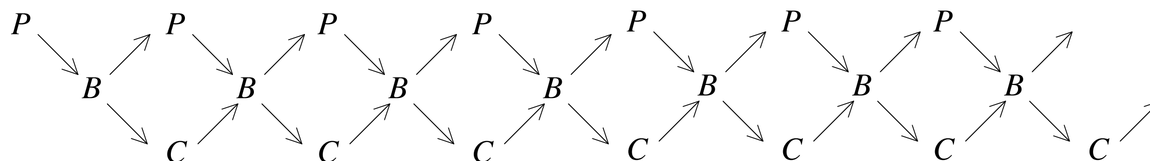
and *newcontrol* can be transformed to

$$newcontrol \;=\; (consume \,\|\, produce).\ newcontrol$$

In this transformed program, the implementation of *consume* will have to capture a copy of the initial value of b. Or, we could do this capture at source level by splitting b into two variables, p and c, as follows.

$$produce \;=\; \cdots\cdots p{:=}\, e\cdots\cdots$$
$$consume \;=\; \cdots\cdots x{:=}\, c\cdots\cdots$$
$$control \;=\; produce.\ newcontrol$$
$$newcontrol \;=\; c{:=}\, p.\ (consume \,\|\, produce).\ newcontrol$$

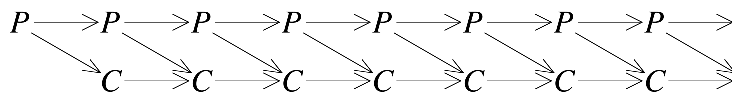
Using B for the assignment $c{:=}\, p$, execution is



If one of *produce* or *consume* consistently takes longer than the other, this is the best that can be done. If their execution times vary so that in some cycles *produce* takes longer while in others *consume* takes longer, we can improve by splitting the buffer into an infinite list. We need natural variable w to indicate how much *produce* has written into the buffer, and natural variable r to indicate how much *consume* has read from the buffer. We initialize both w and r to 0. Then

$$produce \;=\; \cdots\cdots bw{:=}\, e.\ w{:=}\, w{+}1\cdots\cdots$$
$$consume \;=\; \cdots\cdots x{:=}\, br.\ r{:=}\, r{+}1\cdots\cdots$$
$$control \;=\; produce.\ consume.\ control$$

If $w \neq r$ then *produce* and *consume* can be executed in parallel, as follows.



When the execution of *produce* is fast, it can get arbitrarily far ahead of the execution of *consume* . When the execution of *consume* is fast, it can catch up to *produce* but not pass it; the sequence is retained when $w=r$. The opportunity for parallel execution can be found automatically by the programming language implementation (compiler), or it can be told to the implementation in some suitable notation. But, in this example, the resulting execution pattern is not expressible as a source program without additional interactive constructs (Chapter 9).

If the buffer is a finite list of length n , we can use it in a cyclic fashion with this modification:

$produce \;=\; \cdots\cdots bw:= e.\; w:= mod\ (w+1)\ n \cdots\cdots$
$consume \;=\; \cdots\cdots x:= br.\; r:= mod\ (r+1)\ n \cdots\cdots$
$control \;=\; produce.\; consume.\; control$

As before, *consume* cannot overtake *produce* because $w=r$ when the buffer is empty. But now *produce* cannot get more than n executions ahead of *consume* because $w=r$ also when the buffer is full.

End of Buffer

Programs are sometimes easier to develop and prove when they do not include any mention of concurrency. The burden of finding concurrency can be placed upon a clever implementation. Synchronization is what remains of sequential execution after all opportunities for concurrency have been found.

8.1.1 Insertion Sort

Exercise 169 asks for a program to sort a list in time bounded by the square of the length of the list. Here is a solution. Let the list be L , and define

$sort \;=\; \langle n\rightarrow \forall i, j: 0,..n\cdot i\le j \Rightarrow Li \le Lj\rangle$

so that $sort\ n$ says that L is sorted up to index n . The specification is

$(\ L'$ is a permutation of $L\ ) \land sort'\ (\#L) \land t' \le t + (\#L)^2$

We leave the first conjunct informal, and ensure that it is satisfied by making all changes to L using

$swap\ i\ j \;=\; Li:= Lj \,||\, Lj:= Li$

We ignore the last conjunct; program transformation will give us a linear time solution. The second conjunct is equal to $sort\ 0 \Rightarrow sort'\ (\#L)$ since $sort\ 0$ is a theorem.

$sort\ 0 \Rightarrow sort'\ (\#L) \;\Leftarrow\;$ **for** $n:= 0;..\#L$ **do** $sort\ n \Rightarrow sort'\ (n+1)$

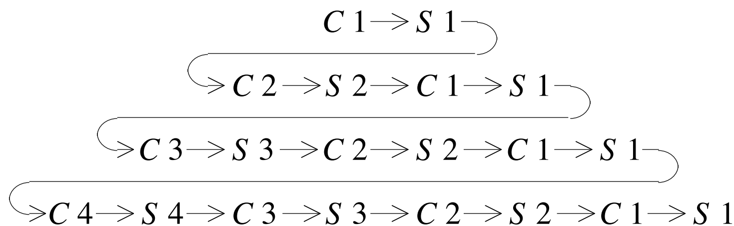
To solve $sort\ n \Rightarrow sort'\ (n+1)$, it may help to refer to an example list.

$$[\ \underset{0}{}\ L0\ \underset{1}{;}\ L1\ \underset{2}{;}\ L2\ \underset{3}{;}\ L3\ \underset{4}{;}\ L4\ \underset{5}{]}$$

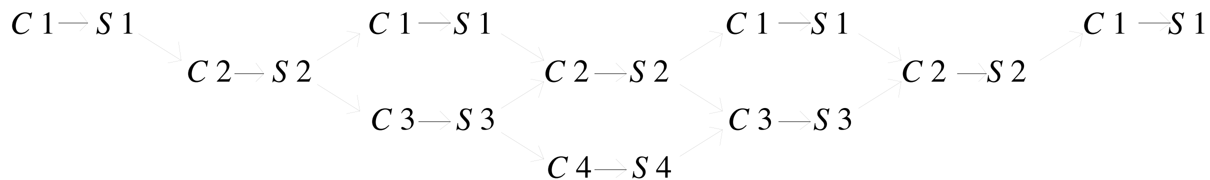
$sort\ n \Rightarrow sort'\ (n+1) \;\Leftarrow\;$ **if** $n=0$ **then** *ok*
else if $L\ (n-1) \le L\ n$ **then** *ok*
else $(swap\ (n-1)\ n.\;\; sort\ (n-1) \Rightarrow sort'\ n)$

If we consider $sort\ n \Rightarrow sort'\ (n+1)$ to be a procedure with parameter n we are finished; the final specification $sort\ (n-1) \Rightarrow sort'\ n$ calls the same procedure with argument $n-1$. Or, we could let n be a variable instead of a **for**-loop index, and decrease it by 1 just before the final call. We leave this detail, and move on to the possibilities for parallel execution.

Let $C\ n$ stand for the comparison $L\ (n-1) \le L\ n$ and let $S\ n$ stand for $swap\ (n-1)\ n$. For $\#L=5$, the worst case sequential execution is shown in the following picture.



If i and j differ by more than 1, then $S\ i$ and $S\ j$ can be executed concurrently. Under the same condition, $S\ i$ can be executed and $C\ j$ can be evaluated concurrently. And of course, any two expressions such as $C\ i$ and $C\ j$ can always be evaluated concurrently. Execution becomes



For the ease of writing a quadratic-time sequential sort, given a clever implementation, we obtain a linear-time parallel sort.

End of Insertion Sort

8.1.2 Dining Philosophers

Exercise 384: Five philosophers are sitting around a round table. At the center of the table is an infinite bowl of noodles. Between each pair of neighboring philosophers is a chopstick. Whenever a philosopher gets hungry, the hungry philosopher reaches for the chopstick on the left and the chopstick on the right, because it takes two chopsticks to eat. If either chopstick is unavailable because the neighboring philosopher is using it, then this hungry philosopher will have to wait until it is available again. When both chopsticks are available, the philosopher eats for a while, then puts down the chopsticks, and goes back to thinking, until the philosopher gets hungry again. The problem is to write a program whose execution simulates the life of these philosophers. It may happen that all five philosophers get hungry at the same time, they each pick up their left chopstick, they then notice that their right chopstick isn't there, and they each decide to wait for their right chopstick while holding on to their left chopstick. That's a deadlock, and the program must be written so that doesn't happen. If we write the program so that only one philosopher gets hungry at a time, there won't be any deadlock, but there won't be much concurrency either.

This problem is a standard one, used in many textbooks, to illustrate the problems of concurrency in programming. There is often one more criterion: each philosopher eats infinitely many times. But we won't bother with that. We'll start with the one-at-a-time version in which there is no concurrency and no deadlock. Number the philosophers from 0 through 4 going round the

table. Likewise number the chopsticks so that the two chopsticks for philosopher i are numbered i and $i+1$ (all additions in this exercise are modulo 5).

$$life = (P\,0 \lor P\,1 \lor P\,2 \lor P\,3 \lor P\,4).\ life$$
$$P\,i = up\ i.\ up(i+1).\ eat\ i.\ down\ i.\ down(i+1)$$
$$up\ i = chopstick\ i := \top$$
$$down\ i = chopstick\ i := \bot$$
$$eat\ i = \cdots\cdots chopstick\ i \cdots\cdots chopstick(i+1) \cdots\cdots$$

These definitions say that life is a completely arbitrary sequence of $P\,i$ actions (choose any one, then repeat), where a $P\,i$ action says that philosopher i picks up the left chopstick, then picks up the right chopstick, then eats, then puts down the left chopstick, then puts down the right chopstick. For these definitions to become a program, we need to decide how to make the choice among the $P\,i$ each iteration; or perhaps we can leave it to the implementation to make the choice (this is where the criterion that each philosopher eats infinitely often would be met). It is unclear how to define $eat\ i$, except that it uses two chopsticks. (If this program were intended to accomplish some purpose, we could eliminate variable $chopstick$, replacing both occurrences in $eat\ i$ by $\top$. But the program is intended to describe an activity, and eating makes use of two chopsticks.)

Now we transform to get concurrency.

If $i \neq j$, $(up\ i.\ up\ j)$ becomes $(up\ i \parallel up\ j)$.
If $i \neq j$, $(up\ i.\ down\ j)$ becomes $(up\ i \parallel down\ j)$.
If $i \neq j$, $(down\ i.\ up\ j)$ becomes $(down\ i \parallel up\ j)$.
If $i \neq j$, $(down\ i.\ down\ j)$ becomes $(down\ i \parallel down\ j)$.
If $i \neq j \land i+1 \neq j$, $(eat\ i.\ up\ j)$ becomes $(eat\ i \parallel up\ j)$.
If $i \neq j \land i \neq j+1$, $(up\ i.\ eat\ j)$ becomes $(up\ i \parallel eat\ j)$.
If $i \neq j \land i+1 \neq j$, $(eat\ i.\ down\ j)$ becomes $(eat\ i \parallel down\ j)$.
If $i \neq j \land i \neq j+1$, $(down\ i.\ eat\ j)$ becomes $(down\ i \parallel eat\ j)$.
If $i \neq j \land i+1 \neq j \land i \neq j+1$, $(eat\ i.\ eat\ j)$ becomes $(eat\ i \parallel eat\ j)$.

Different chopsticks can be picked up or put down at the same time. Eating can be in parallel with picking up or putting down a chopstick, as long as it isn't one of the chopsticks being used for the eating. And finally, two philosophers can eat at the same time as long as they are not neighbors. All these transformations are immediately seen from the definitions of up , $down$, eat , and independent composition. They are not all immediately applicable to the original program, but whenever a transformation is made, it may enable further transformations.

Before any transformation, there is no possibility of deadlock. No transformation introduces the possibility. The result is the maximum concurrency that does not lead to deadlock. A clever implementation can take the initial program (without concurrency) and make the transformations.

A mistake often made in solving the problem of the dining philosophers is to start with too much concurrency.

$$life = P\,0 \parallel P\,1 \parallel P\,2 \parallel P\,3 \parallel P\,4$$
$$P\,i = (up\ i \parallel up(i+1)).\ eat\ i.\ (down\ i \parallel down(i+1)).\ P\,i$$

Clearly $P\,0$ cannot be placed in parallel with $P\,1$ because they both assign and use $chopstick\ 1$. Those who start this way must then try to correct the error by adding mutual exclusion devices and deadlock avoidance devices, and that is what makes the problem hard. It is better not to make the error; then the mutual exclusion devices and deadlock avoidance devices are not needed.

End of Dining Philosophers

End of Sequential to Parallel Transformation

End of Concurrency