9 Interaction

We have been describing computation according to the initial values and final values of state variables. A state variable declaration

var x: T· S = $\exists x, x'$: T· S

says that a state variable is really two mathematical variables, one for the initial value and one for the final value. Within the scope of the declaration, x and x' are available for use in specification S. There are intermediate values whenever there is a dependent (sequential) composition, but these intermediate values are local to the definition of dependent composition.

$P.Q = \exists x'', y'', ...\cdot \langle x', y', ... \rightarrow P\rangle\ x''\ y''\ ...\ \wedge\ \langle x, y, ... \rightarrow Q\rangle\ x''\ y''\ ...$

Consider $(P.Q) \parallel R$. The intermediate values between P and Q are hidden in the dependent composition, and are not visible to R, so they cannot be used for process interaction.

A variable whose value is visible only initially and finally is called a boundary variable, and a variable whose value is visible all the time is called an interactive variable. So far our variables have all been boundary variables. Now we introduce interactive variables whose intermediate values are visible to parallel processes. These variables can be used to describe and reason about interactions between people and computers, and between processes, during the course of a computation.

9.0 Interactive Variables

Let the notation **ivar** x: T· S declare x to be an interactive variable of type T and scope S. It is defined as follows.

ivar x: T· S = $\exists x$: $time \rightarrow T$· S

where $time$ is the domain of time, usually either the extended integers or the extended reals. An interactive variable is a function of time. The value of variable x at time t is $x\,t$.

Suppose a and b are boundary variables, x and y are interactive variables, and t is time. For independent composition we partition all the state variables, boundary and interactive. Suppose a and x belong to P, and b and y belong to Q.

$$\begin{aligned} P \| Q = \exists t_P, t_Q\cdot \quad & \langle t' \rightarrow P\rangle\ t_P\ \wedge\ (\forall t''\cdot\ t_P \le t'' \le t' \Rightarrow x t'' = x(t_P)) \\ \wedge\ & \langle t' \rightarrow Q\rangle\ t_Q\ \wedge\ (\forall t''\cdot\ t_Q \le t'' \le t' \Rightarrow y t'' = y(t_Q)) \\ \wedge\ & t' = max\ t_P\ t_Q \end{aligned}$$

The new part says that when the shorter process is finished, its interactive variables remain unchanged while the longer process is finishing.

Using the same processes and variables as in the previous paragraph, the assignment $x := a+b+x+y$ in process P assigns to variable x the sum of four values. Since a and x are variables of process P, their values are the latest ones assigned to them by process P, or their initial values if process P has not assigned to them. Since b is a boundary variable of process Q, its value, as seen in P, is its initial value, regardless of whether Q has assigned to it. Since y is an interactive variable of process Q, its value, as seen in P, is the latest one assigned to it by process Q, or its initial value if Q has not assigned to it, or unknown if Q is in the middle of assigning to it. Since x is an interactive variable, its new value can be seen in all parallel processes. The expression $a+b+x+y$ is an abuse of notation, since a and b are numbers and x and y are functions from time to numbers; the value being assigned is actually $a+b+xt+yt$, but we omit the argument t when the context makes it clear. We will similarly write x' to mean xt', and x'' to mean xt''.

The definition of ok says that the boundary variables and time are unchanged. So in process P of the previous two paragraphs,

$$ok \;=\; a'=a \;\wedge\; t'=t$$

There is no need to say $x'=x$, which means $xt'=xt$, since $t'=t$. We do not mention b and y because they are not variables of process P.

Assignment to an interactive variable cannot be instantaneous because it is time that distinguishes its values. In a process where the boundary variables are a and b, and the interactive variables are x and y,

$$x{:=}\,e \;=\; a'=a \;\wedge\; b'=b \;\wedge\; x'=e \;\wedge\; (\forall t''\cdot\, t\le t''\le t' \Rightarrow y''=y)$$
$$\wedge\; t' = t+(\text{the time required to evaluate and store } e)$$

interactive variable y remains unchanged throughout the duration of the assignment to x. Nothing is said about the value of x during the assignment.

Assignment to a boundary variable can be instantaneous if we wish. If we choose to account for its time, we must say that all interactive variables remain unchanged during the assignment.

Dependent composition hides the intermediate values of the boundary and time variables, leaving the intermediate values of the interactive variables visible. In boundary variables a and b, and interactive variables x and y, and time t, we define

$$P.Q \;=\; \exists a'', b'', t''\cdot\, \langle a', b', t'\to P\rangle\, a''\; b''\; t'' \;\wedge\; \langle a, b, t\to Q\rangle\, a''\; b''\; t''$$

Most of the specification laws and refinement laws survive the addition of interactive variables, but sadly, the Substitution Law no longer works.

If processes P and Q are in parallel, they have different variables. Suppose again that boundary variable a and interactive variable x are the variables of process P, and that boundary variable b and interactive variable y are the variables of process Q. In specification P, the inputs are a, b, xt, and yt'' for $t\le t''<t'$. In specification P, the outputs are a', and xt'' for $t<t''\le t'$. Specification P is implementable when

$$\forall a, b, X, y, t\cdot\, \exists a', x, t'\cdot\; P \;\wedge\; t\le t' \;\wedge\; \forall t''\cdot\, t<t''\le t' \;\vee\; x\,t''=X\,t''$$

As before, P must be satisfiable with nondecreasing time; the new part says that P must not constrain its interactive variables outside the interval from t to t'. We do not need to know the context of a process specification to check its implementability; variables b and y appear only in the outside universal quantification.

Exercise 385 is an example in the same variables a, b, x, y, and t. Suppose that time is an extended integer, and that each assignment takes time 1.

$(x{:=}\,2.\;\; x{:=}\,x+y.\;\; x{:=}\,x+y) \parallel (y{:=}\,3.\;\; y{:=}\,x+y)$

Clearly, x is a variable in the left process and y is a variable in the right process. Let's put a in the left process and b in the right process.

$$= \quad (a'=a \wedge xt'=2 \wedge t'=t+1.\; a'=a \wedge xt'=xt+yt \wedge t'=t+1.\; a'=a \wedge xt'=xt+yt \wedge t'=t+1)$$
$$\parallel\; (b'=b \wedge yt'=3 \wedge t'=t+1.\;\; b'=b \wedge yt'=xt+yt \wedge t'=t+1)$$
$$= \quad (a'=a \wedge x(t+1)=2 \wedge x(t+2)=x(t+1)+y(t+1) \wedge x(t+3)=x(t+2)+y(t+2) \wedge t'=t+3)$$
$$\parallel\; (b'=b \wedge y(t+1)=3 \wedge y(t+2)=x(t+1)+y(t+1) \wedge t'=t+2)$$
$$= \quad a'=a \wedge x(t+1)=2 \wedge x(t+2)=x(t+1)+y(t+1) \wedge x(t+3)=x(t+2)+y(t+2)$$
$$\wedge\; b'=b \wedge y(t+1)=3 \wedge y(t+2)=x(t+1)+y(t+1) \wedge y(t+3)=y(t+2) \wedge t'=t+3$$
$$= \quad a'=a \wedge x(t+1)=2 \wedge x(t+2)=5 \wedge x(t+3)=10$$
$$\wedge\; b'=b \wedge y(t+1)=3 \wedge y(t+2)=y(t+3)=5 \wedge\; t'=t+3$$

The example gives the appearance of lock-step synchrony only because we took each assignment time to be 1. More realistically, different assignments take different times, perhaps specified nondeterministically with lower and upper bounds. Whatever timing policy we decide on, whether deterministic or nondeterministic, whether discrete or continuous, the definitions and theory remain unchanged. Of course, complicated timing leads quickly to very complicated expressions that describe all possible interactions. If we want to know only something, not everything, about the possible behaviors, we can proceed by implications instead of equations, weakening for the purpose of simplifying. Programming goes the other way: we start with a specification of desired behavior, and strengthen as necessary to obtain a program.

9.0.0 Thermostat

Exercise 388: specify a thermostat for a gas burner. The thermostat operates in parallel with other processes

thermometer || *control* || *thermostat* || *burner*

The thermometer and the control are typically located together, but they are logically distinct. The inputs to the thermostat are:

- real *temperature*, which comes from the thermometer and indicates the actual temperature.
- real *desired*, which comes from the control and indicates the desired temperature.
- boolean *flame*, which comes from a flame sensor in the burner and indicates whether there is a flame.

These three variables must be interactive variables because their values may be changed at any time by another process and the thermostat must react to their current values. These three variables do not belong to the thermostat, and cannot be assigned values by the thermostat. The outputs of the thermostat are:

- boolean *gas*; assigning it $\top$ turns the gas on and $\bot$ turns the gas off.
- boolean *spark*; assigning it $\top$ causes sparks for the purpose of igniting the gas.

Variables *gas* and *spark* belong to the thermostat process. They must also be interactive variables; the burner needs their current values.

Heat is wanted when the actual temperature falls ε below the desired temperature, and not wanted when the actual temperature rises ε above the desired temperature, where ε is small enough to be unnoticeable, but large enough to prevent rapid oscillation. To obtain heat, the spark should be applied to the gas for at least 1 second to give it a chance to ignite and to allow the flame to become stable. But a safety regulation states that the gas must not remain on and unlit for more than 3 seconds. Another regulation says that when the gas is shut off, it must not be turned on again for at least 20 seconds to allow any accumulated gas to clear. And finally, the gas burner must respond to its inputs within 1 second.

Here is a specification:

thermostat = (*gas*:= $\bot$ || *spark*:= $\bot$). *GasIsOff*

GasIsOff = **if** *temperature* < *desired* – ε
then ((*gas*:= $\top$ || *spark*:= $\top$ || t+1 ≤ t' ≤ t+3). *spark*:= $\bot$. *GasIsOn*)
else (((**frame** *gas*, *spark*· *ok*) || $t < t' \le t$+1). *GasIsOff*)

GasIsOn = **if** *temperature* < *desired* + ε ∧ *flame*
then (((**frame** *gas*, *spark*· *ok*) || $t < t' \le t$+1). *GasIsOn*)
else ((*gas*:= $\bot$ || (**frame** *spark*· *ok*) || t+20 ≤ t' ≤ t+21). *GasIsOff*)

We are using the time variable to represent real time in seconds. The specification $t{+}1 \le t' \le t{+}3$ represents the passage of at least 1 second but not more than 3 seconds. The specification $t{+}20 \le t' \le t{+}21$ is similar. A specification that a computation be slow enough is always easy to satisfy. A specification that it be fast enough requires us to build fast enough hardware; in this case it is easy since instruction times are microseconds and the time bounds are seconds.

One can always argue about whether a formal specification captures the intent of an informal specification. For example, if the gas is off, and heat becomes wanted, and the ignition sequence begins, and then heat is no longer wanted, this last input may not be noticed for up to 3 seconds. It may be argued that this is not responding to an input within 1 second, or it may be argued that the entire ignition sequence is the response to the first input, and until its completion no response to further inputs is required. At least the formal specification is unambiguous.

End of Thermostat

9.0.1 Space

The main purpose of interactive variables is to provide a means for processes to interact. In this subsection, we show another use. We make the space variable s into an interactive variable in order to look at the space occupied during the course of a computation. As an example, Exercise 389 is contrived to be as simple as possible while including time and space calculations in an infinite computation.

Suppose *alloc* allocates 1 unit of memory space and takes time 1 to do so. Then the following computation slowly allocates memory.

$$GrowSlow \Leftarrow \textbf{if } t{=}2{\times}x \textbf{ then } (alloc \parallel x{:=}\, t) \textbf{ else } t{:=}\, t{+}1.\ GrowSlow$$

If the time is equal to $2{\times}x$, then one space is allocated, and in parallel x becomes the time stamp of the allocation; otherwise the clock ticks. The process is repeated forever. Prove that if the space is initially less than the logarithm of the time, and x is suitably initialized, then at all times the space is less than the logarithm of the time.

It is not clear what initialization is suitable for x, so leaving that aside for a moment, we define *GrowSlow* to be the desired specification.

$$GrowSlow = s < log\ t \Rightarrow (\forall t''\cdot\ t'' \ge t \Rightarrow s'' < log\ t'')$$

where s is an interactive variable, so s is really $s\ t$ and s'' is really $s\ t''$. We are just interested in the space calculation and not in actually allocating space, so we can take *alloc* to be $s{:=}\, s{+}1$. There is no need for x to be interactive, so let's make it a boundary variable. To make the proof easier, we let all variables be extended naturals, although the result we are proving holds also for real time.

Now we have to prove the refinement, and to do that it helps to break it into pieces. The body of the loop can be written as a disjunction.

$$\begin{array}{lll} & \textbf{if } t{=}2{\times}x \textbf{ then } (s{:=}\, s{+}1 \parallel x{:=}\, t) \textbf{ else } t{:=}\, t{+}1 & \\ = & t{=}2{\times}x \wedge s'{=}s{+}1 \wedge x'{=}t \wedge t'{=}t{+}1 \ \vee\ t{\neq}2{\times}x \wedge s'{=}s \wedge x'{=}x \wedge t'{=}t{+}1 & \end{array}$$

Now the refinement has the form

$$\begin{array}{lll} & (A{\Rightarrow}B \Leftarrow C{\vee}D.\ A{\Rightarrow}B) & .\ \text{distributes over } \vee \\ = & (A{\Rightarrow}B \Leftarrow (C.\ A{\Rightarrow}B) \vee (D.\ A{\Rightarrow}B)) & \text{antidistributive law} \\ = & (A{\Rightarrow}B \Leftarrow (C.\ A{\Rightarrow}B)) \wedge (A{\Rightarrow}B \Leftarrow (D.\ A{\Rightarrow}B)) & \text{portation twice} \\ = & (B \Leftarrow A \wedge (C.\ A{\Rightarrow}B)) \wedge (B \Leftarrow A \wedge (D.\ A{\Rightarrow}B)) & \end{array}$$

So we can break the proof into two cases:

$B \Leftarrow A \wedge (C.\, A \Rightarrow B)$
$B \Leftarrow A \wedge (D.\, A \Rightarrow B)$

starting each time with the right side (antecedent) and working toward the left side (consequent). First case:

$s < \log t \wedge (\, t{=}2{\times}x \wedge s'{=}s{+}1 \wedge x'{=}t \wedge t'{=}t{+}1.$
$\quad s < \log t \Rightarrow \forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t'')$
remove dependent composition, remembering that s is interactive
$= \quad s < \log t \wedge (\exists x'', t'''{\cdot}\quad t{=}2{\times}x \wedge s'''{=}s{+}1 \wedge x''{=}t \wedge t'''{=}t{+}1$
$\quad \wedge\ (s''' < \log t''' \Rightarrow \forall t''{\cdot}\ t''{\geq}t''' \Rightarrow s'' < \log t''))$
Use $s'''{=}s{+}1$ and drop it. Use one-point to eliminate $\exists x'', t'''$.
$\Rightarrow \quad s < \log t \wedge t{=}2{\times}x \wedge (s{+}1 < \log(t{+}1) \Rightarrow \forall t''{\cdot}\ t''{\geq}t{+}1 \Rightarrow s'' < \log t'')$

The next step should be discharge. We need

$s < \log t \wedge t{=}2{\times}x \Rightarrow s{+}1 < \log(t{+}1)$
$= \quad 2^s < t = 2{\times}x \Rightarrow 2^{s+1} < t{+}1$
$= \quad 2^s < t = 2{\times}x \Rightarrow 2^{s+1} \leq t$
$= \quad 2^s < t = 2{\times}x \Rightarrow 2^{s+1} \leq 2{\times}x$
$= \quad 2^s < t = 2{\times}x \Rightarrow 2^s \leq x$
$\Leftarrow \quad 2^s \leq x$

This is the missing initialization of x. So we go back and redefine *GrowSlow*.

GrowSlow $= \ s < \log t \wedge x{\geq}2^s \Rightarrow (\forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t'')$

Now we redo the proof. First case:

$s < \log t \wedge x{\geq}2^s \wedge (\, t{=}2{\times}x \wedge s'{=}s{+}1 \wedge x'{=}t \wedge t'{=}t{+}1.$
$\quad s < \log t \wedge x{\geq}2^s \Rightarrow \forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t'')$
remove dependent composition, remembering that s is interactive
$= \quad s < \log t \wedge x{\geq}2^s$
$\wedge\ (\exists x'', t'''{\cdot}\quad t{=}2{\times}x \wedge s'''{=}s{+}1 \wedge x''{=}t \wedge t'''{=}t{+}1$
$\quad \wedge\ (s''' < \log t''' \wedge x'' {\geq} 2^{s'''} \Rightarrow \forall t''{\cdot}\ t''{\geq}t''' \Rightarrow s'' < \log t''))$
Use $s'''{=}s{+}1$ and drop it. Use one-point to eliminate $\exists x'', t'''$.
$\Rightarrow \quad s < \log t \wedge x{\geq}2^s \wedge t{=}2{\times}x$
$\wedge\ (s{+}1 < \log(t{+}1) \wedge t{\geq}2^{s+1} \Rightarrow \forall t''{\cdot}\ t''{\geq}t{+}1 \Rightarrow s'' < \log t'')$
discharge, as calculated earlier
$= \quad s < \log t \wedge x{\geq}2^s \wedge t{=}2{\times}x \wedge \forall t''{\cdot}\ t''{\geq}t{+}1 \Rightarrow s'' < \log t''$
when $t''{=}t$, then $s''{=}s$ and since $s < \log t$, the domain of t'' can be increased
$\Rightarrow \quad \forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t''$

The second case is easier than the first.

$s < \log t \wedge x{\geq}2^s \wedge (\, t{\neq}2{\times}x \wedge s'{=}s \wedge x'{=}x \wedge t'{=}t{+}1.$
$\quad s < \log t \wedge x{\geq}2^s \Rightarrow \forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t'')$
remove dependent composition, remembering that s is interactive
$= \quad s < \log t \wedge x{\geq}2^s$
$\wedge\ (\exists x'', t'''{\cdot}\quad t{\neq}2{\times}x \wedge s'''{=}s \wedge x''{=}x \wedge t'''{=}t{+}1$
$\quad \wedge\ (s''' < \log t''' \wedge x'' {\geq} 2^{s'''} \Rightarrow \forall t''{\cdot}\ t''{\geq}t''' \Rightarrow s'' < \log t''))$
Use $s'''{=}s$ and drop it. Use one-point to eliminate $\exists x'', t'''$.
$\Rightarrow \quad s < \log t \wedge x{\geq}2^s \wedge t{\neq}2{\times}x$
$\wedge\ (s < \log t \wedge x{\geq}2^s \Rightarrow \forall t''{\cdot}\ t''{\geq}t{+}1 \Rightarrow s'' < \log t'')$ discharge
$= \quad s < \log t \wedge x{\geq}2^s \wedge t{\neq}2{\times}x \wedge \forall t''{\cdot}\ t''{\geq}t{+}1 \Rightarrow s'' < \log t''$
when $t''{=}t$, then $s''{=}s$ and since $s < \log t$, the domain of t'' can be increased
$\Rightarrow \quad \forall t''{\cdot}\ t''{\geq}t \Rightarrow s'' < \log t''$

End of Space

End of Interactive Variables

A shared variable is a variable that can be written and read by any process. Shared variables are popular for process interaction, but they present enormous problems for people who wish to reason about their programs, and for those who must build the hardware and software to implement them. For their trouble, there is no benefit. Interactive variables are not fully shared; all processes can read an interactive variable, but only one process can write it. Interactive variables are easier to reason about and implement than fully shared variables. Even boundary variables are shared a little: their initial values are visible to all processes. They are easiest to reason about and implement, but they provide the least interaction.

Although interactive variables are tamer than shared variables, there are still two problems with them. The first is that they provide too much information. Usually, a process does not need the values of all interactive variables at all times; each process needs only something about the values (an expression in interactive variables), and only at certain times. The other problem is that processes may be executed on different processors, and the rates of execution may not be identical. This makes it hard to know exactly when to read the value of an interactive variable; it certainly should not be read while its owner process is in the middle of writing to it.

We now turn to a form of communication between processes that does not have these problems: it provides just the right information, and mediates the timing between the processes. And, paradoxically, it provides the means for fully sharing variables safely.

9.1 Communication

This section introduces named communication channels through which a computation communicates with its environment, which may be people or other computations running in parallel. For each channel, only one process (person or computation) writes to it, but all processes can read all the messages, each at its own speed. For two-way communication, use two channels. We start the section by considering only one reading process, which may be the same process that writes, or may be a different process. We consider multiple reading processes later when we come to Subsection 9.1.9 on broadcast.

Communication on channel c is described by two infinite strings $\mathbb{M}c$ and $\mathbb{T}c$ called the message script and the time script, and two extended natural variables rc and wc called the read cursor and the write cursor. The message script is the string of all messages, past, present, and future, that pass along the channel. The time script is the corresponding string of times that the messages were or are or will be sent. The scripts are state constants, not state variables. The read cursor is a state variable saying how many messages have been read, or input, on the channel. The write cursor is a state variable saying how many messages have been written, or output, on the channel. If there is only one channel, or if the channel is known from context, we may leave out the channel name, abbreviating the names of the scripts and cursors to $\mathbb{M}$, $\mathbb{T}$, w, and r.

During execution, the read and write cursors increase as inputs and outputs occur; more and more of the script items are seen, but the scripts do not vary. At any time, the future messages and the times they are sent on a channel may be unknown, but they can be referred to as items in the scripts. For example, after 2 more reads the next input on channel c will be $\mathbb{M}c_{\,rc+2}$, and after 5 more writes the next output will be $\mathbb{M}c_{\,wc+5}$ and it will occur at time $\mathbb{T}c_{\,wc+5}$. Omitting the channel name from the script and cursor names, after 2 more reads the next input will be $\mathbb{M}_{r+2}$, and after 5 more writes the next output will be $\mathbb{M}_{w+5}$ at time $\mathbb{T}_{w+5}$.

$$\begin{array}{lll} \mathbb{M} & = & 6\ ;\ 4\ ;\ 7\ ;\ 1\ ;\ 0\ ;\ 3\ ;\ 8\ ;\ 9\ ;\ 2\ ;\ 5\ ;\ ... \\ \mathbb{T} & = & 3\ ;\ 5\ ;\ 5\ ;\ 20\ ;\ 25\ ;\ 28\ ;\ 31\ ;\ 31\ ;\ 45\ ;\ 48\ ;\ ... \\ & & \qquad\qquad \uparrow \qquad\quad \uparrow \\ & & \qquad\qquad r \qquad\quad w \end{array}$$

The scripts and the cursors are not programming notations, but they allow us to specify any desired communications. Here is an example specification. It says that if the next input on channel c is even, then the next output on channel d will be $\top$, and otherwise it will be $\bot$. Formally, we may write

if *even* $(\mathbb{M}c_{rc})$ **then** $\mathbb{M}d_{wd} = \top$ **else** $\mathbb{M}d_{wd} = \bot$

or, more briefly,

$\mathbb{M}d_{wd} = even\ (\mathbb{M}c_{rc})$

If there are only a finite number of communications on a channel, then after the last message, the time script items are all ∞, and the message script items are of no interest.

9.1.0 Implementability

Consider computations involving two memory variables x and y, a time variable t, and communications on a single channel. The state of a computation consists of the values of the memory variables, the time variable, and the cursor variables. During a computation, the memory variables can change value in any direction, but time and the cursors can only increase. Once an input has been read, it cannot be unread; once an output has been written, it cannot be unwritten. Every computation satisfies

$t' \geq t \ \wedge\ r' \geq r \ \wedge\ w' \geq w$

An implementable specification can say what the scripts are in the segment written by a computation, that is the segment $\mathbb{M}_{w;..w'}$ and $\mathbb{T}_{w;..w'}$ between the initial and final values of the write cursor, but it cannot specify the scripts outside this segment. Furthermore, the time script must be monotonic, and all its values in this segment must be in the range from t to t'.

A specification S (in initial state σ, final state σ', message script $\mathbb{M}$, and time script $\mathbb{T}$) is implementable if and only if

$$\begin{array}{ll} \forall \sigma, \mathbb{M}'', \mathbb{T}'' \cdot \exists \sigma', \mathbb{M}, \mathbb{T} \cdot & S \ \wedge\ t' \geq t \ \wedge\ r' \geq r \ \wedge\ w' \geq w \\ & \wedge\ \mathbb{M}_{(0;..w);\ (w';..\infty)} = \mathbb{M}''_{(0;..w);\ (w';..\infty)} \\ & \wedge\ \mathbb{T}_{(0;..w);\ (w';..\infty)} = \mathbb{T}''_{(0;..w);\ (w';..\infty)} \\ & \wedge\ \forall i, j: w,..w' \cdot i \leq j \Rightarrow t \leq \mathbb{T}_i \leq \mathbb{T}_j \leq t' \end{array}$$

If we have many channels, we need similar conjuncts for each. If we have no channels, implementability reduces to the definition given in Chapter 4.

To implement communication channels, it is not necessary to build two infinite strings. At any given time, only those messages that have been written and not yet read need to be stored. The time script is only for specification and proof, and does not need to be stored at all.

End of Implementability

9.1.1 Input and Output

Here are five programming notations for communication. Let c be a channel. The notation $c!\ e$ describes a computation that writes the output message e on channel c. The notation $c!$ describes a computation that sends a signal on channel c (no message; the act of signalling is the only information). The notation $c?$ describes a computation that reads one input on channel c. We use the channel name c to denote the message that was last previously read on the channel. And $\surd c$ is a boolean expression meaning "there is unread input available on channel c". Here are the formal definitions.

$c!\ e \quad = \quad \mathbb{M}_w = e \ \wedge\ \mathbb{T}_w = t \ \wedge\ (w := w+1)$ "c output e"
$c! \quad = \quad \mathbb{T}_w = t \ \wedge\ (w := w+1)$ "c signal"
$c? \quad = \quad r := r+1$ "c input"
$c \quad = \quad \mathbb{M}_{r-1}$
$\surd c \quad = \quad \mathbb{T}_r \le t$ "check c"

Suppose the input channel from a keyboard is named *key*, and the output channel to a screen is named *screen*. Then execution of the program

if $\surd$*key*
then (*key*?. **if** *key*="y" **then** *screen*! "If you wish." **else** *screen*! "Not if you don't want.")
else *screen*! "Well?"

tests if a character of input is available, and if so, reads it and prints some output, which depends on the character read, and if not, prints other output.

Let us refine the specification $\mathbb{M}d_{wd} = even\ (\mathbb{M}c_{rc})$ given earlier.

$\mathbb{M}d_{wd} = even\ (\mathbb{M}c_{rc}) \quad \Leftarrow \quad c?.\ d!\ even\ c$

To prove the refinement, we can rewrite the solution as follows:

$c?.\ d!\ even\ c$
$= \quad rc := rc+1.\ \mathbb{M}d_{wd} = even\ (\mathbb{M}c_{rc-1}) \ \wedge\ \mathbb{T}d_{wd} = t \ \wedge\ (wd := wd+1)$
$= \quad \mathbb{M}d_{wd} = even\ (\mathbb{M}c_{rc}) \ \wedge\ \mathbb{T}d_{wd} = t \ \wedge\ rc' = rc+1 \ \wedge\ wc' = wc \ \wedge\ rd' = rd \ \wedge\ wd' = wd+1$

which implies the problem.

A problem specification should be written as clearly, as understandably, as possible. A programmer refines the problem specification to obtain a solution program, which a computer can execute. In our example, the solution seems more understandable than the problem! Whenever that is the case, we should consider using the program as the problem specification, and then there is no need for refinement.

Our next problem is to read numbers from channel c, and write their doubles on channel d. Ignoring time, the specification can be written

$S \quad = \quad \forall n: nat \cdot\ \mathbb{M}d_{wd+n} = 2 \times \mathbb{M}c_{rc+n}$

We cannot assume that the input and output are the first input and output ever on channels c and d. We can only ask that from now on, starting at the initial read cursor rc and initial write cursor wd, the outputs will be double the inputs. This specification can be refined as follows.

$S \quad \Leftarrow \quad c?.\ d!\ 2{\times}c.\ S$

The proof is:

$c?.\ d!\ 2{\times}c.\ S$
$= \quad rc := rc+1.\ \mathbb{M}d_{wd} = 2 \times \mathbb{M}c_{rc-1} \ \wedge\ (wd := wd+1).\ S$
$= \quad \mathbb{M}d_{wd} = 2 \times \mathbb{M}c_{rc} \ \wedge\ \forall n: nat \cdot\ \mathbb{M}d_{wd+1+n} = 2 \times \mathbb{M}c_{rc+1+n}$
$= \quad \forall n: nat \cdot\ \mathbb{M}d_{wd+n} = 2 \times \mathbb{M}c_{rc+n}$
$= \quad S$

End of Input and Output

9.1.2 Communication Timing

In the real time measure, we need to know how long output takes, how long communication transit takes, and how long input takes, and we place time increments appropriately. To be independent of these implementation details, we can use the transit time measure, in which we suppose that the acts of input and output take no time at all, and that communication transit takes 1 time unit.

The message to be read next on channel c is $\mathbb{M}c_{rc}$. This message was or is or will be sent at time $\mathbb{T}c_{rc}$. Its arrival time, according to the transit time measure, is $\mathbb{T}c_{rc} + 1$. So input becomes

$t := max\ t\ (\mathbb{T}c_{rc} + 1).\ c?$

If the input has already arrived, $\mathbb{T}c_{rc} + 1 \leq t$, and no time is spent waiting for input; otherwise execution of $c?$ is delayed until the input arrives. And the input check $\surd c$ becomes

$\surd c \ = \ \mathbb{T}c_{rc} + 1 \leq t$

In some applications (called "batch processing"), all inputs are available at the start of execution; for these applications, we may as well leave out the time assignments for input, and we have no need for the input check. In other applications (called "process control"), inputs are provided at regular intervals by a physical sampling device; the time script (but not the message script) is known in advance. In still other applications (called "interactive computing"), a human provides inputs at irregular intervals, and we have no way of saying what the time script is. In this case, we have to leave out the waiting times, and just attach a note to our calculation saying that execution time will be increased by any time spent waiting for input.

Exercise 407(a): Let W be "wait for input on channel c and then read it". Formally,

$W \ = \ t := max\ t\ (\mathbb{T}_r + 1).\ c?$

Prove $W \ \Leftarrow$ **if** $\surd c$ **then** $c?$ **else** $(t := t+1.\ \ W)$ assuming time is an extended integer. The significance of this exercise is that input is often implemented in just this way, with a test to see if input is available, and a loop if it is not. Proof:

if $\surd c$ **then** $c?$ **else** $(t := t+1.\ W)$ — replace $\surd c$ and W

$=$ **if** $\mathbb{T}_r + 1 \leq t$ **then** $c?$ **else** $(t := t+1.\ \ t := max\ t\ (\mathbb{T}_r + 1).\ c?)$

$=$ **if** $\mathbb{T}_r + 1 \leq t$ **then** $(t := t.\ \ c?)$ **else** $(t := max\ (t+1)\ (\mathbb{T}_r + 1).\ c?)$

If $\mathbb{T}_r + 1 \leq t$, then $t = max\ t\ (\mathbb{T}_r + 1)$.

If $\mathbb{T}_r + 1 > t$ then $max\ (t+1)\ (\mathbb{T}_r + 1) \ = \ \mathbb{T}_r + 1 \ = \ max\ t\ (\mathbb{T}_r + 1)$.

$=$ **if** $\mathbb{T}_r + 1 \leq t$ **then** $(t := max\ t\ (\mathbb{T}_r + 1).\ \ c?)$ **else** $(t := max\ t\ (\mathbb{T}_r + 1).\ \ c?)$

$=$ W

End of Communication Timing

9.1.3 Recursive Communication

optional; requires Chapter 6

Define dbl by the fixed-point construction (including recursive time but ignoring input waits)

$dbl \ = \ c?.\ \ d!\ 2 \times c.\ \ t := t+1.\ \ dbl$

Regarding dbl as the unknown, this equation has several solutions. The weakest is

$\forall n\colon nat \cdot \mathbb{M}d_{wd+n} = 2 \times \mathbb{M}c_{rc+n} \ \wedge \ \mathbb{T}d_{wd+n} = t+n$

A strongest implementable solution is

$(\forall n\colon nat \cdot \mathbb{M}d_{wd+n} = 2 \times \mathbb{M}c_{rc+n} \ \wedge \ \mathbb{T}d_{wd+n} = t+n)$

$\wedge \ rc' = wd' = t' = \infty \ \wedge \ wc' = wc \ \wedge \ rd' = rd$

The strongest solution is $\bot$. If this fixed-point construction is all we know about dbl, then we cannot say that it is equal to a particular one of the solutions. But we can say this: it refines the weakest solution

$\forall n\colon nat \cdot \mathbb{M}d_{wd+n} = 2 \times \mathbb{M}c_{rc+n} \ \wedge \ \mathbb{T}d_{wd+n} = t+n \ \Leftarrow \ dbl$

and it is refined by the right side of the fixed-point construction

$$dbl \Leftarrow c?.\ d!\ 2{\times}c.\ t{:=}\ t{+}1.\ dbl$$

Thus we can use it to solve problems, and we can execute it.

If we begin recursive construction with

$$dbl_0 = \mathsf{T}$$

we find

$$\begin{aligned} dbl_1 &= c?.\ d!\ 2{\times}c.\ t{:=}\ t{+}1.\ dbl_0 \\ &= rc{:=}\ rc{+}1.\ \mathbb{M}d_{\,wd} = 2 \times \mathbb{M}c_{\,rc-1} \ \wedge\ \mathbb{T}d_{\,wd} = t \ \wedge (wd{:=}\ wd{+}1).\ t{:=}\ t{+}1.\ \mathsf{T} \\ &= \mathbb{M}d_{\,wd} = 2 \times \mathbb{M}c_{\,rc} \ \wedge\ \mathbb{T}d_{\,wd} = t \\ dbl_2 &= c?.\ d!\ 2{\times}c.\ t{:=}\ t{+}1.\ dbl_1 \\ &= rc{:=}\ rc{+}1.\ \mathbb{M}d_{\,wd} = 2 \times \mathbb{M}c_{\,rc-1} \ \wedge\ \mathbb{T}d_{\,wd} = t \ \wedge (wd{:=}\ wd{+}1). \\ &\quad\ t{:=}\ t{+}1.\ \mathbb{M}d_{\,wd} = 2 \times \mathbb{M}c_{\,rc} \ \wedge\ \mathbb{T}d_{\,wd} = t \\ &= \mathbb{M}d_{\,wd} = 2 \times \mathbb{M}c_{\,rc} \ \wedge\ \mathbb{T}d_{\,wd} = t \ \wedge\ \mathbb{M}d_{\,wd+1} = 2{\times}\mathbb{M}c_{\,rc+1} \ \wedge\ \mathbb{T}d_{\,wd+1} = t{+}1 \end{aligned}$$

and so on. The result of the construction

$$dbl_\infty = \forall n{:}\ nat{\cdot}\ \mathbb{M}d_{\,wd+n} = 2 \times \mathbb{M}c_{\,rc+n} \ \wedge\ \mathbb{T}d_{\,wd+n} = t{+}n$$

is the weakest solution of the *dbl* fixed-point construction. If we begin recursive construction with $t' \geq t \ \wedge\ rc' \geq rc \ \wedge\ wc' \geq wc \ \wedge\ rd' \geq rd \ \wedge\ wd' \geq wd$ we get a strongest implementable solution.

End of Recursive Communication

9.1.4 Merge

Merging means reading repeatedly from two or more input channels and writing those inputs onto another channel. The output is an interleaving of the messages from the input channels. The output must be all and only the messages read from the inputs, and it must preserve the order in which they were read on each channel. Infinite merging can be specified formally as follows. Let the input channels be c and d, and the output channel be e. Then

$$merge = (c?.\ e!\ c) \vee (d?.\ e!\ d).\ merge$$

This specification does not state any criterion for choosing between the input channels at each step. To write a merge program, we must decide on a criterion for choosing. We might choose between the input channels based on the value of the inputs or on their arrival times.

Exercise 411(a) (time merge) asks us to choose the first available input at each step. If input is already available on both channels c and d, take either one; if input is available on just one channel, take that one; if input is available on neither channel, wait for the first one and take it (in case of a tie, take either one). Here is the specification.

$$\begin{aligned} timemerge = \quad & (\surd c \ \vee\ \mathbb{T}c_{\,rc} \leq \mathbb{T}d_{\,rd}) \ \wedge\ (c?.\ e!\ c) \\ & \vee\ (\surd d \ \vee\ \mathbb{T}c_{\,rc} \geq \mathbb{T}d_{\,rd}) \ \wedge\ (d?.\ e!\ d). \\ & timemerge \end{aligned}$$

To account for the time spent waiting for input, we should insert $t{:=}\ max\ t\ (\mathbb{T}_r + 1)$ just before each input operation, and for recursive time we should insert $t{:=}\ t{+}1$ before the recursive call.

In Subsection 9.1.2 on Communication Timing we proved that waiting for input can be implemented recursively. Using the same reasoning, we implement *timemerge* as follows.

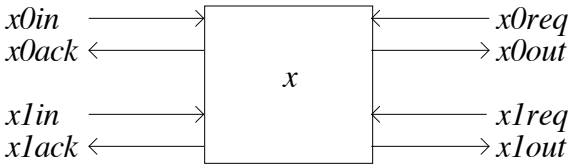
$$\begin{aligned} timemerge \Leftarrow\ & \textbf{if}\ \surd c\ \textbf{then}\ (c?.\ e!\ c)\ \textbf{else}\ ok. \\ & \textbf{if}\ \surd d\ \textbf{then}\ (d?.\ e!\ d)\ \textbf{else}\ ok. \\ & t{:=}\ t{+}1.\ timemerge \end{aligned}$$

assuming time is an extended integer.

End of Merge

9.1.5 Monitor

To obtain the effect of a fully shared variable, we create a process called a monitor that resolves conflicting uses of the variable. A monitor for variable x receives on channels *x0in* , *x1in* , ... data from other processes to be written to the variable, whereupon it sends an acknowledgement back to the writing process on one of the channels *x0ack* , *x1ack* , ... It receives on channels *x0req* , *x1req* , ... requests from other processes to read the variable, whereupon it sends the value of the variable back to the requesting process on one of the channels *x0out* , *x1out* , ...



A monitor for variable x with two writing processes and two reading processes can be defined as follows. Let m be the minimum of the times $\mathbb{T}x0in_{\,rx0in}$, $\mathbb{T}x1in_{\,rx1in}$, $\mathbb{T}x0req_{\,rx0req}$, and $\mathbb{T}x1req_{\,rx1req}$ of the next input on each of the input channels. Then

$$\begin{aligned} monitor \;=\; & (\surd x0in \;\vee\; \mathbb{T}x0in_{\,rx0in} = m) \;\wedge\; (x0in?.\; x := x0in.\; x0ack!) \\ \vee\; & (\surd x1in \;\vee\; \mathbb{T}x1in_{\,rx1in} = m) \;\wedge\; (x1in?.\; x := x1in.\; x1ack!) \\ \vee\; & (\surd x0req \;\vee\; \mathbb{T}x0req_{\,rx0req} = m) \;\wedge\; (x0req?.\; x0out!\; x) \\ \vee\; & (\surd x1req \;\vee\; \mathbb{T}x1req_{\,rx1req} = m) \;\wedge\; (x1req?.\; x1out!\; x). \\ & monitor \end{aligned}$$

Just like *timemerge* , a monitor takes the first available input and responds to it. A monitor for several variables, for several writing processes, and for several reading processes, is similar. When more than one input is available, an implementation must make a choice. Here's one way to implement a monitor, assuming time is an extended integer:

$$\begin{aligned} monitor \;\Leftarrow\; & \textbf{if } \surd x0in \textbf{ then } (x0in?.\; x := x0in.\; x0ack!) \textbf{ else } ok. \\ & \textbf{if } \surd x1in \textbf{ then } (x1in?.\; x := x1in.\; x1ack!) \textbf{ else } ok. \\ & \textbf{if } \surd x0req \textbf{ then } (x0req?.\; x0out!\; x) \textbf{ else } ok. \\ & \textbf{if } \surd x1req \textbf{ then } (x1req?.\; x1out!\; x) \textbf{ else } ok. \\ & t := t{+}1.\; monitor \end{aligned}$$

We earlier solved Exercise 388 to specify a thermostat for a gas burner using interactive variables *gas* , *temperature* , *desired* , *flame* , and *spark* , as follows.

$$thermostat \;=\; (gas := \bot \parallel spark := \bot).\; GasIsOff$$

$$\begin{aligned} GasIsOff \;=\; & \textbf{if } temperature < desired - \varepsilon \\ & \textbf{then } ((gas := \top \parallel spark := \top \parallel t{+}1 \le t' \le t{+}3).\; spark := \bot.\; GasIsOn) \\ & \textbf{else } (((\textbf{frame } gas, spark\cdot\; ok) \parallel t < t' \le t{+}1).\; GasIsOff) \end{aligned}$$

$$\begin{aligned} GasIsOn \;=\; & \textbf{if } temperature < desired + \varepsilon \;\wedge\; flame \\ & \textbf{then } (((\textbf{frame } gas, spark\cdot\; ok) \parallel t < t' \le t{+}1).\; GasIsOn) \\ & \textbf{else } ((gas := \bot \parallel (\textbf{frame } spark\cdot\; ok) \parallel t{+}20 \le t' \le t{+}21).\; GasIsOff) \end{aligned}$$

If we use communication channels instead of interactive variables, we have to build a monitor for these variables, and rewrite our thermostat specification. Here is the result.

$thermostat = ((gasin!\ \bot.\ gasack?) \parallel (sparkin!\ \bot.\ sparkack?)).\ GasIsOff$

$GasIsOff = ((temperaturereq!.\ temperature?) \parallel (desiredreq!.\ desired?)).$
if $temperature < desired - \varepsilon$
then $(((gasin!\ \top.\ gasack?) \parallel (sparkin!\ \top.\ sparkack?) \parallel t{+}1 \le t' \le t{+}3).$
$sparkin!\ \bot.\ sparkack?.\ GasIsOn\ )$
else $(t < t' \le t{+}1.\ GasIsOff)$

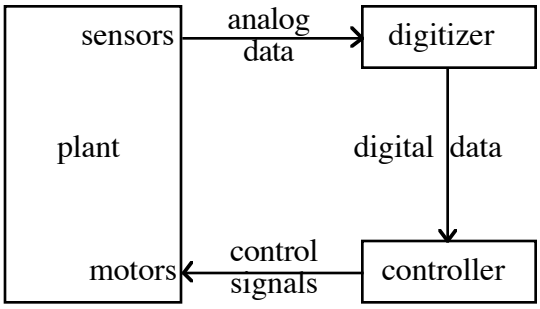
$GasIsOn = (\ (temperaturereq!.\ temperature?) \parallel (desiredreq!.\ desired?)$
$\parallel (flamereq!.\ flame?)\ ).$
if $temperature < desired + \varepsilon\ \wedge\ flame$
then $(t < t' \le t{+}1.\ GasIsOn)$
else $(((gasin!\ \bot.\ gasack?) \parallel t{+}20 \le t' \le t{+}21).\ GasIsOff)$

End of Monitor

The calculation of space requirements when there is concurrency may sometimes require a monitor for the space variable, so that any process can request an update, and the updates can be communicated to all processes. The monitor for the space variable is also the arbiter between competing space allocation requests.

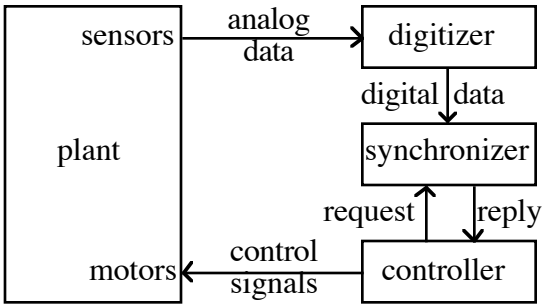
9.1.6 Reaction Controller

Many kinds of reactions are controlled by a feedback loop, as shown in the following picture.



The "plant" could be a chemical reactor, or a nuclear reactor, or even just an assembly plant. The sensors detect concentrations or temperatures or positions in the form of analog data, and feed them to a digitizer. The digitizer converts these data to digital form suitable for the controller. The controller computes what should happen next to control the plant; perhaps some rods should be pushed in farther, or some valves should be opened, or a robot arm should move in some direction. The controller sends signals to the plant to cause the appropriate change.

Here's the problem. The sensors send their data continuously to the digitizer. The digitizer is fast and uniform, sending digital data rapidly to the controller. The time required by the controller to compute its output signals varies according to the input messages; sometimes the computation is trivial and it can keep up with the input; sometimes the computation is more complex and it falls behind. When several inputs have piled up, the controller should not continue to read them and compute outputs in the hope of catching up. Instead, we want all but the latest input to be discarded. It is not essential that control signals be produced as rapidly as digital data. But it is essential that each control signal be based on the latest available data. How can we achieve this? The solution is to place a synchronizer between the digitizer and controller, as in the following picture.



The synchronizer's job is as simple and uniform as the digitizer's; it can easily keep up. It repeatedly reads the data from the digitizer, always keeping only the latest. Whenever the controller requests some data, the synchronizer sends the latest. This is exactly the function of a monitor, and we could implement the synchronizer that way. But a synchronizer is simpler than a monitor in two respects: first, there is only one writing process and one reading process; second, the writing process is uniformly faster than the reading process. Here is its definition.

synchronizer = *digitaldata*?.
if √*request* **then** (*request*? ‖ *reply*! *digitaldata*) **else** *ok*.
synchronizer

If we were using interactive variables instead of channels, there would be no problem of reading old data; reading an interactive variable always reads its latest value, even if the variable is written more often than it is read. But there would be the problem of how to make sure that the interactive variable is not read while it is being written.

End of Reaction Controller

9.1.7 Channel Declaration

The next input on a channel is not necessarily the one that was last previously written on that channel. In one variable x and one channel c (ignoring time),

$c!\ 2.\ c?.\ x := c$
$= \quad \mathbb{M}_w = 2 \ \wedge\ w' = w+1 \ \wedge\ r' = r+1 \ \wedge\ x' = \mathbb{M}_r$

We do not know that initially $w=r$, so we cannot conclude that finally $x'=2$. That's because there may have been a previous write that hasn't been read yet. For example,

$c!\ 1.\ c!\ 2.\ c?.\ x := c$

The next input on a channel is always the first one on that channel that has not yet been read. The same is true in a parallel composition.

$c!\ 2 \ \|\ (c?.\ x := c)$
$= \quad \mathbb{M}_w = 2 \ \wedge\ w' = w+1 \ \wedge\ r' = r+1 \ \wedge\ x' = \mathbb{M}_r$

Again we cannot say $x'=2$ because there may be a previous unread output

$c!\ 1.\ (c!\ 2 \ \|\ (c?.\ x := c)).\ c?$

and the final value of x may be the 1 from the earlier output, with the 2 going to the later input. In order to achieve useful communication between processes, we have to introduce a local channel.

Channel declaration is similar to variable declaration; it defines a new channel within some local portion of a program or specification. A channel declaration applies to what follows it, according to the precedence table on the final page of this book. Here is a syntax and equivalent specification.

chan c: $T \cdot P \quad = \quad \exists \mathbb{M}c$: $\infty * T \cdot \exists \mathbb{T}c$: $\infty * xreal \cdot$ **var** rc , wc: $xnat := 0 \cdot \ P$

The type T says what communications are possible on this new channel. The declaration introduces two scripts, which are infinite strings; they are not state variables, but state constants of

unknown value (mathematical variables). We have let time be extended real, but we could let it be extended integer. The channel declaration also introduces a read cursor rc with initial value 0 to say that initially there has been no input on this channel, and a write cursor wc with initial value 0 to say that initially there has been no output on this channel.

A local channel can be used without concurrency as a queue, or buffer. For example,

chan c: $int\cdot$ $c!\,3.\ c!\,4.\ c?.\ x := c.\ c?.\ x := x+c$

assigns 7 to x. Here is the proof, including time.

chan c: $int\cdot$ $c!\,3.\ c!\,4.\ t := max\ t\ (\mathbb{T}_r + 1).\ c?.\ x := c.\ t := max\ t\ (\mathbb{T}_r + 1).\ c?.\ x := x+c$

$=$ $\exists \mathbb{M}$: $\infty^* int\cdot\ \exists \mathbb{T}$: $\infty^* xint\cdot$ **var** r, w: $xnat := 0\cdot$
$\mathbb{M}_w = 3\ \wedge\ \mathbb{T}_w = t\ \wedge\ (w := w+1).$
$\mathbb{M}_w = 4\ \wedge\ \mathbb{T}_w = t\ \wedge\ (w := w+1).$
$t := max\ t\ (\mathbb{T}_r + 1).\ r := r+1.$
$x := \mathbb{M}_{r-1}.$
$t := max\ t\ (\mathbb{T}_r + 1).\ r := r+1.$
$x := x + \mathbb{M}_{r-1}$

now use the Substitution Law several times

$=$ $\exists \mathbb{M}$: $\infty^* int\cdot\ \exists \mathbb{T}$: $\infty^* xint\cdot\ \exists r, r', w, w'$: $xnat\cdot$
$\mathbb{M}_0 = 3\ \wedge\ \mathbb{T}_0 = t\ \wedge\ \mathbb{M}_1 = 4\ \wedge\ \mathbb{T}_1 = t\ \wedge\ r' = 2\ \wedge\ w' = 2\ \wedge\ x' = \mathbb{M}_0 + \mathbb{M}_1$
$\wedge\ t' = max\ (max\ t\ (\mathbb{T}_0 + 1))\ (\mathbb{T}_1 + 1)\ \wedge$ (other variables unchanged)

$=$ $x'=7\ \wedge\ t' = t+1\ \wedge$ (other variables unchanged)

Here are two processes with a communication between them. Ignoring time,

chan c: $int\cdot$ $c!\,2\ \|\ (c?.\ x := c)$ Use the definition of local channel declaration, and use the previous result for the independent composition

$=$ $\exists \mathbb{M}$: $\infty^* int\cdot$ **var** r, w: $xnat := 0\cdot$
$\mathbb{M}_w = 2\ \wedge\ w' = w+1\ \wedge\ r' := r+1\ \wedge\ x' = \mathbb{M}_r\ \wedge$ (other variables unchanged)

Now apply the initialization $r := 0$ and $w := 0$ using the Substitution Law

$=$ $\exists \mathbb{M}$: $\infty^* int\cdot$ **var** r, w: $xnat\cdot$
$\mathbb{M}_0 = 2\ \wedge\ w'=1\ \wedge\ r'=1\ \wedge\ x' = \mathbb{M}_0\ \wedge$ (other variables unchanged)

$=$ $x'=2\ \wedge$ (other variables unchanged)

$=$ $x := 2$

Replacing 2 by an arbitrary expression, we have a general theorem equating communication on a local channel with assignment. If we had included time, the result would have been

$x'=2\ \wedge\ t' = t+1\ \wedge$ (other variables unchanged)

$=$ $x := 2.\ t := t+1$

End of Channel Declaration

9.1.8 Deadlock

In the previous subsection we saw that a local channel can be used as a buffer. Let's see what happens if we try to read first and write after. Inserting the input wait into

chan c: $int\cdot$ $c?.\ c!\,5$

gives us

chan c: int· t:= $max\ t\ (\mathbb{T}_r + 1)$. c?. c! 5

$=$ $\exists \mathbb{M}$: ∞*int· $\exists \mathbb{T}$: ∞*$xint$· **var** r, w: $xnat := 0$·
t:= $max\ t\ (\mathbb{T}_r + 1)$. r:= r+1. $\mathbb{M}_w = 5\ \wedge\ \mathbb{T}_w = t\ \wedge (w := w+1)$

We'll do this one slowly. First, expand **var** and w:= w+1 , taking r , w , x , and t as the state variables.

$=$ $\exists \mathbb{M}$: ∞*int· $\exists \mathbb{T}$: ∞*$xint$· $\exists r, r', w, w'$: $xnat$·
$r := 0$. $w := 0$. $t := max\ t\ (\mathbb{T}_r + 1)$. $r := r+1$.
$\mathbb{M}_w = 5\ \wedge\ \mathbb{T}_w = t\ \wedge\ r'=r\ \wedge\ w' = w+1\ \wedge\ x'=x\ \wedge\ t'=t$

Now use the Substitution Law four times.

$=$ $\exists \mathbb{M}$: ∞*int· $\exists \mathbb{T}$: ∞*$xint$· $\exists r, r', w, w'$: $xnat$·
$\mathbb{M}_0 = 5\ \wedge\ \mathbb{T}_0 = max\ t\ (\mathbb{T}_0 + 1)\ \wedge\ r'=1\ \wedge\ w'=1 \wedge x'=x\ \wedge t' = max\ t\ (\mathbb{T}_0 + 1)$

Look at the conjunct $\mathbb{T}_0 = max\ t\ (\mathbb{T}_0 + 1)$. For any start time $t > -\infty$ it says $\mathbb{T}_0 = \infty$.

$=$ $x'=x\ \wedge\ t'=\infty$

The theory tells us that execution takes forever because the wait for input is infinite.

The word "deadlock" is usually used to mean that several processes are waiting on each other, as in the dining philosophers example of Chapter 8. But it might also be used to mean that a single sequential computation is waiting on itself, as in the previous paragraph. Here's the more traditional example with two processes.

chan c, d: int· $(c?.\ d!\ 6)\ \|\ (d?.\ c!\ 7)$

Inserting the input waits, we get

chan c, d: int· $(t := max\ t\ (\mathbb{T}c_{\,rc} + 1).\ c?.\ d!\ 6)\ \|\ (t := max\ t\ (\mathbb{T}d_{\,rd} + 1).\ d?.\ c!\ 7)$

after a little work, we obtain

$=$ $\exists \mathbb{M}c, \mathbb{M}d$: ∞*int· $\exists \mathbb{T}c, \mathbb{T}d$: ∞*$xint$· $\exists rc, rc', wc, wc', rd, rd', wd, wd'$: $xnat$·
$\mathbb{M}d_{\,0} = 6\ \wedge\ \mathbb{T}d_{\,0} = max\ t\ (\mathbb{T}c_{\,0} + 1)\ \wedge\ \mathbb{M}c_{\,0} = 7\ \wedge\ \mathbb{T}c_{\,0} = max\ t\ (\mathbb{T}d_{\,0} + 1)$
$\wedge\ \ rc'=wc'=rd'=wd'=1 \wedge x'=x\ \wedge\ t' = max\ (max\ t\ (\mathbb{T}c_{\,0} + 1))\ (max\ t\ (\mathbb{T}d_{\,0} + 1))$

Once again, for start time $t > -\infty$, the conjuncts $\mathbb{T}d_{\,0} = max\ t\ (\mathbb{T}c_{\,0} + 1)\ \wedge\ \mathbb{T}c_{\,0} = max\ t\ (\mathbb{T}d_{\,0} + 1)$ tell us that $\mathbb{T}d_{\,0} = \mathbb{T}c_{\,0} = \infty$.

$=$ $x'=x\ \wedge\ t'=\infty$

To prove that a computation is free from deadlock, prove that all message times are finite.

End of Deadlock

9.1.9 Broadcast

A channel consists of a message script, a time script, a read cursor, and a write cursor. Whenever a computation splits into parallel processes, the state variables must be partitioned among the processes. The scripts are not state variables; they do not belong to any process. The cursors are state variables, so one of the processes can write to the channel, and one (perhaps the same one, perhaps a different one) can read from the channel. Suppose the structure is

$P.\ (Q\ \|\ R\ \|\ S).\ T$

and suppose Q writes to channel c and R reads from channel c . The messages written by Q follow those written by P , and those written by T follow those written by Q . The messages read by R follow those read by P , and those read by T follow those read by R . There is no problem of two processes attempting to write at the same time, and the timing discipline makes sure that reading a message waits until after it is written.

Although communication on a channel, as defined so far, is one-way from a single writer to a single reader, we can have as many channels as we want. So we can have two-way conversations between

all pairs of processes. But sometimes it is convenient to have a broadcast from one process to more than one of the parallel processes. In the program structure of the previous paragraph, we might want Q to write and both of R and S to read on the same channel. Broadcast is achieved by several read cursors, one for each reading process. Then all reading processes read the same messages, each at its own rate. There is no harm in two processes reading the same message, even at the same time. But there is a problem with broadcast: which of the read cursors becomes the read cursor for T? All of the read cursors start with the same value, but they may not end with the same value. There is no sensible way to continue reading from that channel. So we allow broadcast on a channel only when the parallel composition is not followed sequentially by a program that reads from that channel.

We next present a broadcast example that combines communicating processes, local channel declaration, and dynamic process generation, in one beautiful little program. It is also a striking example of the importance of good notation and good theory. It has been "solved" before without them, but the "solutions" required many pages, intricate synchronization arguments, lacked proof, and were sometimes wrong.

Exercise 415 is multiplication of power series: Write a program to read from channel a an infinite sequence of coefficients $a_0\ a_1\ a_2\ a_3\ ...$ of a power series $a_0 + a_1{\times}x + a_2{\times}x^2 + a_3{\times}x^3 + ...$ and in parallel to read from channel b an infinite sequence of coefficients $b_0\ b_1\ b_2\ b_3\ ...$ of a power series $b_0 + b_1{\times}x + b_2{\times}x^2 + b_3{\times}x^3 + ...$ and in parallel to write on channel c the infinite sequence of coefficients $c_0\ c_1\ c_2\ c_3\ ...$ of the power series $c_0 + c_1{\times}x + c_2{\times}x^2 + c_3{\times}x^3 + ...$ equal to the product of the two input series. Assume that all inputs are already available; there are no input delays. Produce the outputs one per time unit.

The question provides us with a notation for the coefficients: $a_n = \mathbb{M}a_{\,ra+n}$, $b_n = \mathbb{M}b_{\,rb+n}$, and $c_n = \mathbb{M}c_{\,rc+n}$. Let us use A, B, and C for the power series, so we can express our desired result as

$$\begin{aligned} C &= A{\times}B \\ &= (a_0 + a_1{\times}x + a_2{\times}x^2 + a_3{\times}x^3 + ...) \times (b_0 + b_1{\times}x + b_2{\times}x^2 + b_3{\times}x^3 + ...) \\ &= a_0{\times}b_0 + (a_0{\times}b_1 + a_1{\times}b_0){\times}x + (a_0{\times}b_2 + a_1{\times}b_1 + a_2{\times}b_0){\times}x^2 \\ &\quad + (a_0{\times}b_3 + a_1{\times}b_2 + a_2{\times}b_1 + a_3{\times}b_0){\times}x^3 + ... \end{aligned}$$

from which we see $c_n = \Sigma i{:}\ 0{,}..n{+}1{\cdot}\ a_i{\times}b_{n-i}$. The question relieves us from concern with input times, but we are still concerned with output times. The complete specification is

$$C = A{\times}B \ \wedge\ \forall n{\cdot}\ \mathbb{T}c_{\,wc+n} = t{+}n$$

Consider the problem: output coefficient n requires $n{+}1$ multiplications and n additions from $2{\times}(n{+}1)$ input coefficients, and it must be produced 1 time unit after the previous coefficient. To accomplish this requires more and more data storage, and more and more parallelism, as execution progresses.

As usual, let us concentrate on the result first, and leave the time for later. Let

$$A_1 = a_1 + a_2{\times}x + a_3{\times}x^2 + a_4{\times}x^3 + ...$$
$$B_1 = b_1 + b_2{\times}x + b_3{\times}x^2 + b_4{\times}x^3 + ...$$

be the power series from channels a and b beginning with coefficient 1. Then

$$\begin{aligned} & A{\times}B \\ = \ & (a_0 + A_1{\times}x) \times (b_0 + B_1{\times}x) \\ = \ & a_0{\times}b_0 + (a_0{\times}B_1 + A_1{\times}b_0){\times}x + A_1{\times}B_1{\times}x^2 \end{aligned}$$

In place of the problem $A\times B$ we have five new problems. The first is to read one coefficient from each input channel and output their product; that's easy. The next two, $a_0\times B_1$ and $A_1\times b_0$, are multiplying a power series by a constant; that's easier than multiplying two power series, requiring only a loop. The next, $A_1\times B_1$, is exactly the problem we started with, but one coefficient farther along; it can be solved by recursion. Finally, we have to add three power series together. Unfortunately, these three power series are not synchronized properly. We must add the leading coefficients of $a_0\times B_1$ and $A_1\times b_0$ without any coefficient from $A_1\times B_1$, and thereafter add coefficient $n+1$ of $a_0\times B_1$ and $A_1\times b_0$ to coefficient n of $A_1\times B_1$. To synchronize, we move $a_0\times B_1$ and $A_1\times b_0$ one coefficient farther along. Let

$A_2 = a_2 + a_3\times x + a_4\times x^2 + a_5\times x^3 + ...$

$B_2 = b_2 + b_3\times x + b_4\times x^2 + b_5\times x^3 + ...$

be the power series from channels a and b beginning with coefficient 2. Continuing the earlier equation for $A\times B$,

$= \quad a_0\times b_0 + (a_0\times(b_1 + B_2\times x) + (a_1 + A_2\times x)\times b_0)\times x + A_1\times B_1\times x^2$

$= \quad a_0\times b_0 + (a_0\times b_1 + a_1\times b_0)\times x + (a_0\times B_2 + A_1\times B_1 + A_2\times b_0)\times x^2$

From this expansion of the desired product we can almost write a solution directly.

One problem remains. A recursive call will be used to obtain a sequence of coefficients of the product $A_1\times B_1$ in order to produce the coefficients of $A\times B$. But the output channel for $A_1\times B_1$ cannot be channel c, the output channel for the main computation $A\times B$. Instead, a local channel must be used for output from $A_1\times B_1$. We need a channel parameter, for which we invent the notation $\langle!\,\rangle$. A channel parameter is really four parameters: one for the message script, one for the time script, one for the write cursor, and one for the read cursor. (The cursors are variables, so their parameters are reference parameters; see Subsection 5.5.2.)

Now we are ready. Define P (for product) to be our specification (ignoring time for a moment) parameterized by output channel.

$P = \langle!c\colon rat\rightarrow C = A\times B\rangle$

We refine $P\,c$ as follows.

$$P\,c \Leftarrow (a?\parallel b?).\ c!\ a\times b.$$
$$\textbf{var}\ a0\colon rat := a\cdot\ \textbf{var}\ b0\colon rat := b\cdot\ \textbf{chan}\ d\colon rat\cdot$$
$$P\,d \parallel ((a?\parallel b?).\ c!\ a0\times b + a\times b0.\ C = a0\times B + D + A\times b0)$$

$$C = a0\times B + D + A\times b0 \Leftarrow (a?\parallel b?\parallel d?).\ c!\ a0\times b + d + a\times b0.\ C = a0\times B + D + A\times b0$$

That is the whole program: 4 lines! First, an input is read from each of channels a and b and their product is output on channel c; that takes care of $a_0\times b_0$. We will need these values again, so we declare local variables (really constants) $a0$ and $b0$ to retain their values. Now that we have read one message from each input channel, we call $P\,d$ to provide the coefficients of $A_1\times B_1$ on local channel d, in parallel with the remainder of the program. Both $P\,d$ and its parallel process will be reading from channels a and b using separate read cursors; there is no computation sequentially following them. In parallel with $P\,d$ we read the next inputs a_1 and b_1 and output the coefficient $a_0\times b_1 + a_1\times b_0$. Finally we execute the loop specified as $C = a0\times B + D + A\times b0$, where D is the power series whose coefficients are read from channel d.

The proof is completely straightforward. Here it is in detail. We start with the right side of the first refinement.

$(a? \parallel b?).\ c!\ a{\times}b.$
var $a0$: $rat := a\cdot$ **var** $b0$: $rat := b\cdot$ **chan** d: $rat\cdot$
$P\ d \parallel ((a? \parallel b?).\ c!\ a0{\times}b + a{\times}b0.\ C = a0{\times}B + D + A{\times}b0)$

$=$ $(ra := ra{+}1 \parallel rb := rb{+}1).\ \mathbb{M}c_{\,wc} = \mathbb{M}a_{\,ra-1} \times \mathbb{M}b_{\,rb-1} \ \wedge\ (wc := wc{+}1).$
$\exists a0, a0', b0, b0', \mathbb{M}d, rd, rd', wd, wd'\cdot$
$a0 := \mathbb{M}a_{\,ra-1}.\ b0 := \mathbb{M}b_{\,rb-1}.\ rd := 0.\ wd := 0.$
$(\forall n\cdot \mathbb{M}d_{\,wd+n} = (\Sigma i{:}\ 0,..n{+}1\cdot \mathbb{M}a_{\,ra+i} \times \mathbb{M}b_{\,rb+n-i}))$
$\wedge\ ((ra := ra{+}1 \parallel rb := rb{+}1).\ \mathbb{M}c_{\,wc} = a0{\times}\mathbb{M}b_{\,rb-1} + \mathbb{M}a_{\,ra-1}{\times}b0 \wedge (wc := wc{+}1).$
$\forall n\cdot \mathbb{M}c_{\,wc+n} = a0 \times \mathbb{M}b_{\,rb+n} + \mathbb{M}d_{\,rd+n} + \mathbb{M}a_{\,ra+n} \times b0)$
Make all substitutions indicated by assignments.

$=$ $\mathbb{M}c_{\,wc} = \mathbb{M}a_{\,ra} \times \mathbb{M}b_{\,rb}$
$\wedge\ \exists a0, a0', b0, b0', \mathbb{M}d, rd, rd', wd, w_d'\cdot$
$(\forall n\cdot \mathbb{M}d_{\,n} = \Sigma i{:}\ 0,..n{+}1\cdot \mathbb{M}a_{\,ra+1+i} \times \mathbb{M}b_{\,rb+1+n-i})$
$\wedge\ \mathbb{M}c_{\,wc+1} = \mathbb{M}a_{\,ra} \times \mathbb{M}b_{\,rb+1} + \mathbb{M}a_{\,ra+1} \times \mathbb{M}b_{\,rb}$
$\wedge\ (\forall n\cdot \mathbb{M}c_{\,wc+2+n} = \mathbb{M}a_{\,ra} \times \mathbb{M}b_{\,rb+2+n} + \mathbb{M}d_{\,n} + \mathbb{M}a_{\,ra+2+n} \times \mathbb{M}b_{\,rb})$
Use the first universal quantification to replace $\mathbb{M}_d n$ in the second.
Then throw away the first universal quantification (weakening our expression).
Now all existential quantifications are unused, and can be thrown away.

$\Rightarrow$ $\mathbb{M}c_{\,wc} = \mathbb{M}a_{\,ra} \times \mathbb{M}b_{\,rb}$
$\wedge\ \mathbb{M}c_{\,wc+1} = \mathbb{M}a_{\,ra} \times \mathbb{M}b_{\,rb+1} + \mathbb{M}a_{\,ra+1} \times \mathbb{M}b_{\,rb}$
$\wedge\ \forall n\cdot \mathbb{M}c_{\,wc+2+n} = \mathbb{M}a_{\,ra} \times \mathbb{M}b_{\,rb+2+n}$
$+\ (\Sigma i{:}\ 0,..n{+}1\cdot \mathbb{M}a_{\,ra+1+i} \times \mathbb{M}b_{\,rb+1+n-i})$
$+\ \mathbb{M}a_{\,ra+2+n} \times \mathbb{M}b_{\,rb}$
Now put the three conjuncts together.

$=$ $\forall n\cdot \mathbb{M}c_{\,wc+n} = \Sigma i{:}\ 0,..n{+}1\cdot \mathbb{M}a_{\,ra+i} \times \mathbb{M}b_{\,rb+n-i}$
$=$ $P\ c$

We still have to prove the loop refinement.
$(a? \parallel b? \parallel d?).\ c!\ a0{\times}b + d + a{\times}b0.\ C = a0{\times}B + D + A{\times}b0$

$=$ $(ra := ra{+}1 \parallel rb := rb{+}1 \parallel rd := rd{+}1).$
$\mathbb{M}c_{\,wc} = a0 \times \mathbb{M}b_{\,rb-1} + \mathbb{M}d_{\,rd-1} + \mathbb{M}a_{\,ra-1} \times b0 \ \wedge\ (wc := wc{+}1).$
$\forall n\cdot \mathbb{M}c_{\,wc+n} = a0 \times \mathbb{M}b_{\,rb+n} + \mathbb{M}d_{\,rd+n} + \mathbb{M}a_{\,ra+n} \times b0$
Make all substitutions indicated by assignments.

$=$ $\mathbb{M}c_{\,wc} = a0 \times \mathbb{M}b_{\,rb} + \mathbb{M}d_{\,rd} + \mathbb{M}a_{\,ra} \times b0$
$\wedge\ \forall n\cdot \mathbb{M}c_{\,wc+1+n} = a0 \times \mathbb{M}b_{\,rb+1+n} + \mathbb{M}d_{\,rd+1+n} + \mathbb{M}a_{\,ra+1+n} \times b0$
Put the two conjuncts together.

$=$ $\forall n\cdot \mathbb{M}c_{\,wc+n} = a0 \times \mathbb{M}b_{\,rb+n} + \mathbb{M}d_{\,rd+n} + \mathbb{M}a_{\,ra+n} \times b0$
$=$ $C = a0{\times}B + D + A{\times}b0$

According to the recursive measure of time, we must place a time increment before the recursive call $P\ d$ and before the recursive call $C = a0{\times}B + D + A{\times}b0$. We do not need a time increment before inputs on channels a and b according to information given in the question. We do need a time increment before the input on channel d. Placing only these necessary time increments, output $c_0 = a_0{\times}b_0$ will occur at time $t{+}0$ as desired, but output $c_1 = a_0{\times}b_1 + a_1{\times}b_0$ will also occur at time $t{+}0$, which is too soon. In order to make output c_1 occur at time $t{+}1$ as desired, we must place a time increment between the first two outputs. We can consider this time increment to account for actual computing time, or as a delay (see Section 5.3, "Time and Space Dependence"). Here is the program with time.

$$Q\ c \Leftarrow (a? \parallel b?).\ c!\ a{\times}b.$$
var $a0$: $rat := a\cdot$ **var** $b0$: $rat := b\cdot$ **chan** d: $rat\cdot$
$(t{:=}\ t{+}1.\ Q\ d) \parallel ((a? \parallel b?).\ t{:=}\ t{+}1.\ c!\ a0{\times}b + a{\times}b0.\ R)$

$$R \Leftarrow (a? \parallel b? \parallel (t{:=}\ max\ t\ (\mathbb{T}d_{\,rd} + 1).\ d?)).\ c!\ a0{\times}b + d + a{\times}b0.\ t{:=}\ t{+}1.\ R$$

where Q and R are defined, as follows:

$$Q\ c = \forall n\cdot \mathbb{T}c_{\,wc+n} = t{+}n$$
$$Q\ d = \forall n\cdot \mathbb{T}d_{\,wd+n} = t{+}n$$
$$R = (\forall n\cdot \mathbb{T}d_{\,rd+n} = t{+}n) \Rightarrow (\forall n\cdot \mathbb{T}c_{\,wc+n} = t{+}1{+}n)$$

Within loop R, the assignment $t{:=}\ max\ t\ (\mathbb{T}d_{\,rd} + 1)$ represents a delay of 1 time unit the first iteration (because $t = \mathbb{T}d_{\,rd}$), and a delay of 0 time units each subsequent iteration (because $t = \mathbb{T}d_{\,rd} + 1$). This makes the proof very ugly. To make the proof pretty, we can replace $t{:=}\ max\ t\ (\mathbb{T}d_{\,rd} + 1)$ by $t{:=}\ max\ (t{+}1)\ (\mathbb{T}d_{\,rd} + 1)$ and delete $t{:=}\ t{+}1$ just before the call to R. These changes together do not change the timing at all; they just make the proof easier. The assignment $t{:=}\ max\ (t{+}1)\ (\mathbb{T}d_{\,rd} + 1)$ increases the time by at least 1, so the loop includes a time increase without the $t{:=}\ t{+}1$. The program with time is now

$$Q\ c \Leftarrow (a? \parallel b?).\ c!\ a{\times}b.$$
var $a0$: $rat := a\cdot$ **var** $b0$: $rat := b\cdot$ **chan** d: $rat\cdot$
$(t{:=}\ t{+}1.\ Q\ d) \parallel ((a? \parallel b?).\ t{:=}\ t{+}1.\ c!\ a0{\times}b + a{\times}b0.\ R)$

$$R \Leftarrow (a? \parallel b? \parallel (t{:=}\ max\ (t{+}1)\ (\mathbb{T}d_{\,rd} + 1).\ d?)).\ c!\ a0{\times}b + d + a{\times}b0.\ R$$

Here is the proof of the first of these refinements, beginning with the right side.

$(a? \parallel b?).\ c!\ a{\times}b.$
var $a0$: $rat := a\cdot$ **var** $b0$: $rat := b\cdot$ **chan** d: $rat\cdot$
$(t{:=}\ t{+}1.\ Q\ d) \parallel ((a? \parallel b?).\ t{:=}\ t{+}1.\ c!\ a0{\times}b + a{\times}b0.\ R)$

We can ignore $a?$ and $b?$ because they have no effect on timing (they are substitutions for variables that do not appear in $Q\ d$ and R). We also ignore what messages are output, looking only at their times. We can therefore also ignore variables $a0$ and $b0$.

$\Rightarrow$ $\mathbb{T}c_{\,wc} = t \ \wedge\ (wc{:=}\ wc{+}1).$
$\exists \mathbb{T}d, rd, rd', wd, wd'\cdot\ rd{:=}\ 0.\ wd{:=}\ 0.$
$(t{:=}\ t{+}1.\ \forall n\cdot \mathbb{T}d_{\,wd+n} = t{+}n)$
$\wedge$ $(\ t{:=}\ t{+}1.\ \mathbb{T}c_{\,wc} = t \ \wedge\ (wc{:=}\ wc{+}1).$
$(\forall n\cdot \mathbb{T}d_{\,rd+n} = t{+}n) \Rightarrow (\forall n\cdot \mathbb{T}c_{\,wc+n} = t{+}1{+}n)\ )$

Make all substitutions indicated by assignments.

$=$ $\mathbb{T}c_{\,wc} = t$
$\wedge$ $\exists \mathbb{T}d, rd, rd', wd, wd'\cdot$
$(\forall n\cdot \mathbb{T}d_{\,n} = t{+}1{+}n)$
$\wedge\ \mathbb{T}c_{\,wc+1} = t{+}1$
$\wedge\ ((\forall n\cdot \mathbb{T}d_{\,n} = t{+}1{+}n) \Rightarrow (\forall n\cdot \mathbb{T}c_{\,wc+2+n} = t{+}2{+}n))$

Use the first universal quantification to discharge the antecedent.
Then throw away the first universal quantification (weakening our expression).
Now all existential quantifications are unused, and can be thrown away.

$\Rightarrow$ $\mathbb{T}c_{\,wc} = t \ \wedge\ \mathbb{T}c_{\,wc+1} = t{+}1 \ \wedge\ \forall n\cdot \mathbb{T}c_{\,wc+2+n} = t{+}2{+}n$

Now put the three conjuncts together.

$=$ $\forall n\cdot \mathbb{T}c_{\,wc+n} = t{+}n$
$=$ $Q\ c$

We still have to prove the loop refinement.

$(R \Leftarrow (a? \parallel b? \parallel (t{:=}\ max\ (t{+}1)\ (\mathbb{T}d_{\,rd} + 1).\ d?)).\ c!\ a0{\times}b + d + a{\times}b0.\ R)$

Ignore $a?$ and $b?$ and the output message.

$\Leftarrow \quad ((\forall n{\cdot}\ \mathbb{T}d_{\,rd+n} = t{+}n) \ \Rightarrow\ (\forall n{\cdot}\ \mathbb{T}c_{\,wc+n} = t{+}1{+}n))$

$\Leftarrow\ (\ t{:=}\ max\ (t{+}1)\ (\mathbb{T}d_{\,rd} + 1).\ rd{:=}\ rd{+}1.\ \mathbb{T}c_{\,wc} = t\ \wedge\ (wc{:=}\ wc{+}1).$
$(\forall n{\cdot}\ \mathbb{T}d_{\,rd+n} = t{+}n) \ \Rightarrow\ (\forall n{\cdot}\ \mathbb{T}c_{\,wc+n} = t{+}1{+}n)\ )$

Use the Law of Portation to move the first antecedent to the right side, where it becomes a conjunct.

$= \quad (\forall n{\cdot}\ \mathbb{T}c_{\,wc+n} = t{+}1{+}n)$

$\Leftarrow \quad (\forall n{\cdot}\ \mathbb{T}d_{\,rd+n} = t{+}n)$

$\wedge\ (\ t{:=}\ max\ (t{+}1)\ (\mathbb{T}d_{\,rd} + 1).\ rd{:=}\ rd{+}1.\ \mathbb{T}c_{\,wc} = t\ \wedge\ (wc{:=}\ wc{+}1).$
$(\forall n{\cdot}\ \mathbb{T}d_{\,rd+n} = t{+}n) \ \Rightarrow\ (\forall n{\cdot}\ \mathbb{T}c_{\,wc+n} = t{+}1{+}n)\ )$

Specializing $\forall n{\cdot}\ \mathbb{T}d_{\,rd+n} = t{+}n$ to the case $n{=}0$, we use $\mathbb{T}d_{\,rd} = t$ to simplify $max\ (t{+}1)\ (\mathbb{T}d_{\,rd} + 1)$.

$= \quad (\forall n{\cdot}\ \mathbb{T}c_{\,wc+n} = t{+}1{+}n)$

$\Leftarrow \quad (\forall n{\cdot}\ \mathbb{T}d_{\,rd+n} = t{+}n)$

$\wedge\ (\ t{:=}\ t{+}1.\ rd{:=}\ rd{+}1.\ \mathbb{T}c_{\,wc} = t\ \wedge\ (wc{:=}\ wc{+}1).$
$(\forall n{\cdot}\ \mathbb{T}d_{\,rd+n} = t{+}n) \ \Rightarrow\ (\forall n{\cdot}\ \mathbb{T}c_{\,wc+n} = t{+}1{+}n)\ )$

Make all substitutions indicated by assignments.

$= \quad (\forall n{\cdot}\ \mathbb{T}c_{\,wc+n} = t{+}1{+}n)$

$\Leftarrow \quad (\forall n{\cdot}\ \mathbb{T}d_{\,rd+n} = t{+}n)$

$\wedge\ \mathbb{T}c_{\,wc} = t{+}1$

$\wedge\ ((\forall n{\cdot}\ \mathbb{T}d_{\,rd+1+n} = t{+}1{+}n) \ \Rightarrow\ (\forall n{\cdot}\ \mathbb{T}c_{\,wc+1+n} = t{+}2{+}n))$

The conjunct $\forall n{\cdot}\ \mathbb{T}d_{\,rd+n} = t{+}n$ discharges the antecedent $\forall n{\cdot}\ \mathbb{T}d_{\,rd+1+n} = t{+}1{+}n$ which can be dropped.

$\Leftarrow \quad (\forall n{\cdot}\ \mathbb{T}c_{\,wc+n} = t{+}1{+}n)$

$\Leftarrow\ \mathbb{T}c_{\,wc} = t{+}1\ \wedge\ (\forall n{\cdot}\ \mathbb{T}c_{\,wc+1+n} = t{+}2{+}n)$

$= \quad \mathsf{T}$

End of Broadcast

End of Communication

End of Interaction

For many students, the first understanding of programs they are taught is how programs are executed. And for many students, that is the only understanding they are given. With that understanding, the only method available for checking whether a program is correct is to test it by executing it with a variety of inputs to see if the resulting outputs are right. All programs should be tested, but there are two problems with testing.

One problem with testing is: how do you know if the outputs are right? Some programs give answers you do not already know (that is why you wrote the program), and testing the program does not tell you if it is right. In that case, you should test to see at least if the answers are reasonable. For other programs, for example, graphics programs for producing pretty pictures, the only way to know if the output is right is to test the program and judge the result.

The other problem with testing is: you cannot try all inputs. Even if all the test cases you try give reasonable answers, there may be errors lurking in untried cases.

If you have read and understood this book to here, you now have an understanding of programs that is completely different from execution. When you prove that a program refines a specification, you are considering all inputs at once, and you are proving that the outputs have the properties stated in the specification. That is far more than can ever be accomplished by testing. But it is also more work than trying some inputs and looking at the outputs. That raises the question: when is the extra assurance of correctness worth the extra work?

If the program you are writing is easy enough that you can probably get it right without any theory, and it does not really matter if there are some errors in it, then the extra assurance of correctness provided by the theory may not be worth the trouble. If you are writing a pacemaker controller for a heart, or the software that controls a subway system, or an air traffic control program, or nuclear power plant software, or any other programs that people's lives will depend on, then the extra assurance is definitely worth the trouble, and you would be negligent if you did not use the theory.

To prove that a program refines a specification after the program is finished is a very difficult task. It is much easier to perform the proof while the program is being written. The information needed to make one step in programming is exactly the same information that is needed to prove that step is correct. The extra work is mainly to write down that information formally. It is also the same information that will be needed later for program modification, so writing it explicitly at each step will save effort later. And if you find, by trying to prove a step, that the step is incorrect, you save the effort of building the rest of your program on a wrong step. As a further bonus, after you become practiced and skillful at using the theory, you find that it helps in the program design; it suggests programming steps. In the end, it may not be any extra effort at all.

In this book we have looked only at small programs. But the theory is not limited to small programs; it is independent of scale, applicable to any size of software. In a large software project, the first design decision might be to divide the task into several pieces that will fit together in some way. This decision can be written as a refinement, specifying exactly what the parts are and how they fit together, and then the refinement can be proven. Using the theory in the early stages is enormously beneficial, because if an early step is wrong, it is enormously costly to correct later.

For a theory of programming to be in widespread use for industrial program design, it must be supported by tools. Ideally, an automated prover checks each refinement, remaining silent if the refinement is correct, complaining whenever there is a mistake, and saying exactly what is wrong. At present there are a few tools that provide some assistance, but they are far from ideal. There is plenty of opportunity for tool builders, and they need a thorough knowledge of a practical theory of programming.

10 Exercises

Exercises marked with √ have been done in previous chapters.

10.0 Preface

0 There are four cards on a table showing symbols D, E, 2, and 3 (one per card). Each card has a letter on one side and a digit on the other. Which card(s) do you need to turn over to determine whether every card with a D on one side has a 3 on the other? Why?

End of Preface

10.1 Basic Theories

1 Simplify each of the following boolean expressions.

(a) $x \wedge \neg x$

(b) $x \vee \neg x$

(c) $x \Rightarrow \neg x$

(d) $x \Leftarrow \neg x$

(e) $x = \neg x$

(f) $x \neq \neg x$

2 Prove each of the following laws of Boolean Theory using the proof format given in Subsection 1.0.1, and any laws listed in Section 11.4. Do not use the Completion Rule.

(a) $a \wedge b \Rightarrow a \vee b$

(b) $(a \wedge b) \vee (b \wedge c) \vee (a \wedge c) \;=\; (a \vee b) \wedge (b \vee c) \wedge (a \vee c)$

(c) $\neg a \Rightarrow (a \Rightarrow b)$

(d) $a = (b \Rightarrow a) \;=\; a \vee b$

(e) $a = (a \Rightarrow b) \;=\; a \wedge b$

(f) $(a \Rightarrow c) \wedge (b \Rightarrow \neg c) \Rightarrow \neg(a \wedge b)$

(g) $a \wedge \neg b \Rightarrow a \vee b$

(h) $(a \Rightarrow b) \wedge (c \Rightarrow d) \wedge (a \vee c) \Rightarrow (b \vee d)$

(i) $a \wedge \neg a \Rightarrow b$

(j) $(a \Rightarrow b) \vee (b \Rightarrow a)$

(k)√ $\neg(a \wedge \neg(a \vee b))$

(l) $(\neg a \Rightarrow \neg b) \wedge (a \neq b) \;\vee\; (a \wedge c \Rightarrow b \wedge c)$

(m) $(a \Rightarrow \neg a) \Rightarrow \neg a$

(n) $(a \Rightarrow b) \wedge (\neg a \Rightarrow b) \;=\; b$

(o) $(a \Rightarrow b) \Rightarrow a \;=\; a$

(p) $a = b \vee a = c \vee b = c$

(q) $a \wedge b \vee a \wedge \neg b \;=\; a$

(r) $a \Rightarrow (b \Rightarrow a)$

(s) $a \Rightarrow a \wedge b \;=\; a \Rightarrow b \;=\; a \vee b \Rightarrow b$

(t) **if** a **then** a **else** $\neg a$

(u) **if** $b \wedge c$ **then** P **else** Q $=$ **if** b **then if** c **then** P **else** Q **else** Q

(v) **if** $b \vee c$ **then** P **else** Q $=$ **if** b **then** P **else if** c **then** P **else** Q

(w) **if** b **then** P **else if** b **then** Q **else** R $=$ **if** b **then** P **else** R

(x) **if if** b **then** c **else** d **then** P **else** Q
$=$ **if** b **then if** c **then** P **else** Q **else if** d **then** P **else** Q

(y) **if** b **then if** c **then** P **else** R **else if** c **then** Q **else** R
$=$ **if** c **then if** b **then** P **else** Q **else** R

3 (dual) One operator is the dual of another operator if it negates the result when applied to the negated operands. The zero-operand operators $\top$ and $\bot$ are each other's duals. If $op_0\,(\neg a) = \neg(op_1\ a)$ then op_0 and op_1 are duals. If $(\neg a)\ op_0\ (\neg b) = \neg(a\ op_1\ b)$ then op_0 and op_1 are duals. And so on for more operands.

(a) Of the 4 one-operand boolean operators, there is 1 pair of duals, and 2 operators that are their own duals. Find them.

(b) Of the 16 two-operand boolean operators, there are 6 pairs of duals, and 4 operators that are their own duals. Find them.

(c) What is the dual of the three-operand operator **if then else** ? Express it using only the operator **if then else** .

(d) The dual of a boolean expression without variables is formed as follows: replace each operator with its dual, adding parentheses if necessary to maintain the precedence. Explain why the dual of a theorem is an antitheorem, and vice versa.

(e) Let P be a boolean expression without variables. From part (d) we know that every boolean expression without variables of the form

$$(\text{dual of } P) = \neg P$$

is a theorem. Therefore, to find the dual of a boolean expression with variables, we must replace each operator by its dual and negate each variable. For example, if a and b are boolean variables, then the dual of $a \wedge b$ is $\neg a \vee \neg b$. And since

$$(\text{dual of } a \wedge b) = \neg(a \wedge b)$$

we have one of the Duality Laws:

$$\neg a \vee \neg b = \neg(a \wedge b)$$

The other of the Duality Laws is obtained by equating the dual and negation of $a \vee b$. Obtain five laws that do not appear in this book by equating a dual with a negation.

(f) Dual operators have truth tables that are each other's vertical mirror reflections. For example, the truth table for $\wedge$ (below left) is the vertical mirror reflection of the truth table for $\vee$ (below right).

$\wedge$:		
$\top\ \top$	$\top$
$\top\ \bot$	$\bot$
$\bot\ \top$	$\bot$
$\bot\ \bot$	$\bot$

$\vee$:		
$\top\ \top$	$\top$
$\top\ \bot$	$\top$
$\bot\ \top$	$\top$
$\bot\ \bot$	$\bot$

Design symbols (you may redesign existing symbols where necessary) for the 4 one-operand and 16 two-operand boolean operators according to the following criteria.

(i) Dual operators should have symbols that are vertical mirror reflections (like $\wedge$ and $\vee$). This implies that self-dual operators have vertically symmetric symbols, and all others have vertically asymmetric symbols.

(ii) If $a\ op_0\ b = b\ op_1\ a$ then op_0 and op_1 should have symbols that are horizontal mirror reflections (like $\Rightarrow$ and $\Leftarrow$). This implies that symmetric operators have horizontally symmetric symbols, and all others have horizontally asymmetric symbols.

4 Truth tables and the Evaluation Rule can be replaced by a new proof rule and some new axioms. The new proof rule says: "A boolean expression does not gain, lose, or change classification when a theorem within it is replaced by another theorem. Similarly, a boolean expression does not gain, lose, or change classification when an antitheorem within it is replaced by another antitheorem.". The truth tables become new axioms; for example, one truth table entry becomes the axiom $\top \vee \top$ and another becomes the axiom $\top \vee \bot$. These two axioms can be reduced to one axiom by the introduction of a variable, giving $\top \vee x$. Write the truth tables as axioms and antiaxioms as succinctly as possible.

5 Complete the following laws of Boolean Theory

(a) $\top =$

(b) $\bot =$

(c) $\neg a =$

(d) $a \wedge b =$

(e) $a \vee b =$

(f) $a = b =$

(g) $a \neq b =$

(h) $a \Rightarrow b =$

by adding a right side using only the following symbols (in any quantity)

(i) $\neg \wedge a\ b\ (\ )$

(ii) $\neg \vee a\ b\ (\ )$

(iii) $\neg \Rightarrow a\ b\ (\ )$

(iv) $\neq \Rightarrow a\ b\ (\ )$

(v) $\neg$ **if then else** $a\ b\ (\ )$

6 (BDD) A BDD (Binary Decision Diagram) is a boolean expression that has one of the following 3 forms: $\top$, $\bot$, **if** variable **then** BDD **else** BDD . For example,

if x **then if** a **then** $\top$ **else** $\bot$ **else if** y **then if** b **then** $\top$ **else** $\bot$ **else** $\bot$

is a BDD. An OBDD (Ordered BDD) is a BDD with an ordering on the variables, and in each **if then else** , the variable in the **if**-part must come before any of the variables in its **then**- and **else**-parts ("before" means according to the ordering). For example, using alphabetic ordering for the variables, the previous example is not an OBDD, but

if a **then if** c **then** $\top$ **else** $\bot$ **else if** b **then if** c **then** $\top$ **else** $\bot$ **else** $\bot$

is an OBDD. An LBDD (Labeled BDD) is a set of definitions of the following 3 forms:

label = $\top$

label = $\bot$

label = **if** variable **then** label **else** label

The labels are separate from the variables; each label used in a **then**-part or **else**-part must be defined by one of the definitions; exactly one label must be defined but unused. The following is an LBDD.

true = $\top$

false = $\bot$

alice = **if** b **then** true **else** false

bob = **if** a **then** alice **else** false

An LOBDD is an LBDD that becomes an OBDD when the labels are expanded. The ordering prevents any recursive use of the labels. The previous example is an LOBDD. An RBDD (Reduced BDD) is a BDD such that, in each **if then else** , the **then**- and **else**-parts differ. An ROBDD is both reduced and ordered; an RLBDD is both reduced and labeled; an RLOBDD is reduced, labeled, and ordered. The previous example is an RLOBDD.

(a) Express $\neg a$, $a \wedge b$, $a \vee b$, $a \Rightarrow b$, $a = b$, $a \neq b$, and **if** a **then** b **else** c as BDDs.

(b) How can you conjoin two OBDDs and get an OBDD?

(c) How can you determine if two RLOBDDs are equal?

(d) How can we represent an RLOBDD in order to determine efficiently if an assignment of values to variables satisfies it (solves it, gives it value $\top$)?

7 Express formally and succinctly that exactly one of three statements is true.

8 Design symbols for the 10 two-operand boolean operators that are not presented in Chapter 1, and find laws about these operators.

9 The Case Analysis Laws equate the three-operand operator **if** *a* **then** *b* **else** *c* to expressions using only two-operand and one-operand operators. In each, the variable *a* appears twice. Find an equal expression using only two-operand and one-operand operators in which the variable *a* appears only once. Hint: use continuing operators.

10 Consider a fully parenthesized expression containing only the symbols ⊤ ⊥ = ≠ ( ) in any quantity and any syntactically acceptable order.

(a) Show that all syntactically acceptable rearrangements are equivalent.

(b) Show that it is equivalent to any expression obtained from it by making an even number of the following substitutions: ⊤ for ⊥, ⊥ for ⊤, = for ≠, ≠ for =.

11 Let *p* and *q* be boolean expressions. Suppose *p* is both a theorem and an antitheorem (the theory is inconsistent).

(a) Prove, using the rules of proof presented, that *q* is both a theorem and an antitheorem.

(b) Is *q*=*q* a theorem or an antitheorem?

12 Formalize each of the following statements as a boolean expression. Start by staying as close as possible to the English, then simplify as much as possible (sometimes no simplification is possible). You will have to introduce new basic boolean expressions like (the door can be opened) for the parts that cannot make use of boolean operators, but for words like "only if" you should use boolean operators. You translate meanings from words to boolean symbols; the meaning of the words may depend on their context and even on facts not explicitly stated. Formalization is not a simple substitution of symbols for words.

(a) The door can only be opened if the elevator is stopped.

(b) Neither the elevator door nor the floor door will open unless both of them do.

(c) Either the motor is jammed or the control is broken.

(d) Either the light is on or it is off.

(e) If you press the button, the elevator will come.

(f) If the power switch is on, the system is operating.

(g) Where there's smoke, there's fire; and there's no smoke; so there's no fire.

(h) Where there's smoke, there's fire; and there's no fire; so there's no smoke.

(i) You can't score if you don't shoot.

(j) If you have a key, only then can you open the door.

(k) No pain, no gain.

(l) No shirt? No shoes? No service!

(m) If it happens, it happens.

13 Formalize each of the following statements. For each pair, either prove they are equivalent or prove they differ.

(a) Don't drink and drive.

(b) If you drink, don't drive.

(c) If you drive, don't drink.

(d) Don't drink and don't drive.

(e) Don't drink or don't drive.

14 Formalize and prove the following argument. If it is raining and Jane does not have her umbrella with her, then she is getting wet. It is raining. Jane is not getting wet. Therefore Jane has her umbrella with her.

15 A sign says:

NO PARKING
7-9am
4-6pm
Mon-Fri

Using variable *t* for time of day and *d* for day, write a boolean expression that says when there is no parking.

16 (tennis) An advertisement for a tennis magazine says "If I'm not playing tennis, I'm watching tennis. And if I'm not watching tennis, I'm reading about tennis.". Assuming the speaker cannot do more than one of these activities at a time,

(a) prove that the speaker is not reading about tennis.

(b) what is the speaker doing?

17 (maid and butler) The maid said she saw the butler in the living room. The living room adjoins the kitchen. The shot was fired in the kitchen, and could be heard in all nearby rooms. The butler, who had good hearing, said he did not hear the shot. Given these facts, prove that someone lied. Use the following abbreviations.

mtt = (the maid told the truth)
btt = (the butler told the truth)
blr = (the butler was in the living room)
bnk = (the butler was near the kitchen)
bhs = (the butler heard the shot)

18 (knights and knaves) There are three inhabitants of an island, named P, Q, and R. Each is either a knight or a knave. Knights always tell the truth. Knaves always lie. For each of the following, write the given information formally, and then answer the questions, with proof.

(a) You ask P: "Are you a knight?". P replies: "If I am a knight, I'll eat my hat.". Does P eat his hat?

(b) P says: "If Q is a knight, then I am a knave.". What are P and Q?

(c) P says: "There is gold on this island if and only if I am a knight.". Can it be determined whether P is a knight or a knave? Can it be determined whether there is gold on the island?

(d) P, Q, and R are standing together. You ask P: "Are you a knight or a knave?". P mumbles his reply, and you don't hear it. So you ask Q: "What did P say?". Q replies: "P said that he is a knave.". Then R says: "Don't believe Q, he's lying.". What are Q and R?

(e) You ask P: "How many of you are knights?". P mumbles. So Q says: "P said there is exactly one knight among us.". R says: "Don't believe Q, he's lying.". What are Q and R?

(f) P says: "We're all knaves.". Q says: "No, exactly one of us is a knight.". What are P, Q, and R?

19 Islands X and Y contain knights who always tell the truth, knaves who always lie, and possibly also some normal people who sometimes tell the truth and sometimes lie. There is gold on at least one of the islands, and the people know which island(s) it is on. You find a message from the pirate who buried the gold, with the following clue (which we take as an axiom): "If there are any normal people on these islands, then there is gold on both islands.". You are allowed to dig on only one island, and you are allowed to ask one question of one random person. What should you ask in order to find out which island to dig on?

20 (caskets) The princess had two caskets, one gold and one silver. Into one she placed her portrait and into the other she placed a dagger. On the gold casket she wrote the inscription: the portrait is not in here. On the silver casket she wrote the inscription: exactly one of these inscriptions is true. She explained to her suitor that each inscription is either true or false (not both), but on the basis of the inscriptions he must choose a casket. If he chooses the one with the portrait, he can marry her; if he chooses the one with the dagger, he must kill himself. Assuming marriage is preferable to death, which casket should he choose?

21 (the unexpected egg) There are two boxes, one red and one blue. One box has an egg in it; the other is empty. You are to look first in the red box, then if necessary in the blue box, to find the egg. But you will not know which box the egg is in until you open the box and see the egg. You reason as follows: "If I look in the red box and find it empty, I'll know that the egg is in the blue box without opening it. But I was told that I would not know which box the egg is in until I open the box and see the egg. So it can't be in the blue box. Now I know it must be in the red box without opening the red box. But again, that's ruled out, so it isn't in either box.". Having ruled out both boxes, you open them and find the egg in one unexpectedly, as originally stated. Formalize the given statements and the reasoning, and thus explain the paradox.

22 A number can be written as a sequence of decimal digits. For the sake of generality, let us consider using the sequence notation with arbitrary expressions, not just digits. For example, $1(2+3)4$ could be allowed, and be equal to 154. What changes are needed to the number axioms?

23 (scale) There is a tradition in programming languages to use a scale operator, e, in the limited context of digit sequences. Thus $12e3$ is equal to 12×10^3. For the sake of generality, let us consider using the scale notation with arbitrary expressions, not just digits. For example, $(6+6)e(5-2)$ could be allowed, and be equal to $12e3$. What changes are needed to the number axioms?

24 When we defined number expressions, we included complex numbers such as $(-1)^{1/2}$, not because we particularly wanted them, but because it was easier than excluding them. If we were interested in complex numbers, we would find that the number axioms given in Subsection 11.4.2 do not allow us to prove many things we might like to prove. For example, we cannot prove $(-1)^{1/2}\times 0 = 0$. How can the axioms be made strong enough to prove things about complex numbers, but weak enough to leave room for ∞?

25 Express formally

(a) the absolute value of a real number x.

(b) the sign of a real number x, which is -1, 0, or $+1$ depending on whether x is negative, zero, or positive.

26 Prove $-\infty<y<\infty \wedge y\neq 0 \Rightarrow (x/y=z = x=z\times y)$.

27 Show that the number axioms become inconsistent when we add the axiom

$$-\infty<y<\infty \Rightarrow x/y\times y = x$$

28 (circular numbers) Redesign the axioms for the extended number system to make it circular, so that $+\infty = -\infty$. Be careful with the transitivity of $<$.

29 Is there any harm in adding the axiom $0/0=5$ to Number Theory?

30 (bracket algebra) Here is a new way to write boolean expressions. An expression can be empty; in other words, nothing is already an expression. If you put a pair of parentheses around an expression, you get another expression. If you put two expressions next to each other, you get another expression. For example,

$$()(())((())())$$

is an expression. The empty expression is bracket algebra's way of writing T; putting parentheses around an expression is bracket algebra's way of negating it, and putting expressions next to each other is bracket algebra's way of conjoining them. So the example expression is bracket algebra's way of saying

$$\neg\mathsf{T}\wedge\neg\neg\mathsf{T}\wedge\neg(\neg\neg\mathsf{T}\wedge\neg\mathsf{T})$$

We can also have variables anywhere in a bracket expression. There are three rules of bracket algebra. If x, y, and z are any bracket expressions, then

$((x))$	can replace or be replaced by	x	double negation rule
$x()y$	can replace or be replaced by	$()$	base rule
$x\,y\,z$	can replace or be replaced by	$x'\,y\,z'$	context rule

where x' is x with occurrences of y added or deleted, and similarly z' is z with occurrences of y added or deleted. The context rule does not say how many occurrences of y are added or deleted; it could be any number from none to all of them. To prove, you just follow the rules until the expression disappears. For example,

	$((a)b((a)b))$	context rule: empty for x, $(a)b$ for y, $((a)b)$ for z
becomes	$((a)b(\quad))$	base rule: $(a)b$ for x and empty for y
becomes	$(\quad(\quad))$	double negation rule: empty for x
becomes		

Since the last expression is empty, all the expressions are proven.

(a) Rewrite the boolean expression

$$\neg(\neg(a\wedge b)\wedge\neg(\neg a\wedge b)\wedge\neg(a\wedge\neg b)\wedge\neg(\neg a\wedge\neg b))$$

as a bracket expression, and then prove it following the rules of bracket algebra.

(b) As directly as possible, rewrite the boolean expression

$$(\neg a\Rightarrow\neg b)\ \wedge\ (a\neq b)\ \vee\ (a\wedge c\Rightarrow b\wedge c)$$

as a bracket expression, and then prove it following the rules of bracket algebra.

(c) Can all boolean expressions be rewritten reasonably directly as bracket expressions?

(d) Can $x\,y$ become $y\,x$ using the rules of bracket algebra?

(e) Can all theorems of boolean algebra, rewritten reasonably directly as bracket expressions, be proven using the rules of bracket algebra?

(f) We interpret empty as T, parentheses as negation, and juxtaposition as conjunction. Is there any other consistent way to interpret the symbols and rules of bracket algebra?

31 Let $\bullet$ be a two-operand infix operator (let's give it precedence 3) whose operands and result are of some type T. Let $\lozenge$ be a two-operand infix operator (let's give it precedence 7) whose operands are of type T and whose result is boolean, defined by the axiom

$$a\lozenge b \;=\; a\bullet b = a$$

(a) Prove if $\bullet$ is idempotent then $\lozenge$ is reflexive.

(b) Prove if $\bullet$ is associative then $\lozenge$ is transitive.

(c) Prove if $\bullet$ is symmetric then $\lozenge$ is antisymmetric.

(d) If T is the booleans and $\bullet$ is $\wedge$, what is $\lozenge$?

(e) If T is the booleans and $\bullet$ is $\vee$, what is $\lozenge$?

(f) If T is the natural numbers and $\lozenge$ is $\leq$, what is $\bullet$?

(g) The axiom defines $\lozenge$ in terms of $\bullet$. Can it be inverted, so that $\bullet$ is defined in terms of $\lozenge$?

32 (family theory) Design a theory of personal relationships. Invent person expressions such as *Jack* , *Jill* , *father of p* , *mother of p* . Invent boolean expressions that use person expressions, such as *p is male* , *p is female* , *p is a parent of q* , *p is a son of q* , *p is a daughter of q* , *p is a child of q* , *p is married to q* , *p=q* . Invent axioms such as (*p is male*) ≠ (*p is female*) . Formulate and prove an interesting theorem.

End of Basic Theories

10.2 Basic Data Structures

33 Simplify

(a) $(1, 7–3) + 4 – (2, 6, 8)$

(b) $nat \times nat$

(c) $nat – nat$

(d) $(nat+1)\times(nat+1)$

34 Prove $\neg\ 7: null$.

35 We defined bunch *null* with the axiom $null: A$. Is there any harm in defining bunch *all* with the axiom $A: all$?

36 Let A be a bunch of booleans such that $A = \neg A$. What is A ?

37 Show that some of the axioms of Bunch Theory listed in Section 2.0 are provable from the other axioms. How many of the axioms can you remove without losing any theorems?

38 (hyperbunch) A hyperbunch is like a bunch except that each element can occur a number of times other than just zero times (absent) or one time (present). The order of elements remains insignificant. (A hyperbunch does not have a characteristic predicate, but a characteristic function with numeric result.) Design notations and axioms for each of the following kinds of hyperbunch.

(a) multibunch: an element can occur any natural number of times. For example, a multibunch can consist of one 2, two 7s, three 5s, and zero of everything else. (Note: the equivalent for sets is called either a multiset or a bag.)

(b) wholebunch: an element can occur any integer number of times.

(c) fuzzybunch: an element can occur any real number of times from 0 to 1 inclusive.

39 A composite number is a natural number with 2 or more (not necessarily distinct) prime factors. Express the composite numbers as simply as you can.

40 For this question only, let # be a two-operand infix operator (precedence 3) with natural operands and an extended natural result. Informally, $n\#m$ means "the number of times that n is a factor of m ". It is defined by the following two axioms.

$m: n\times nat \ \vee\ n\#m = 0$

$n\neq 0 \ \Rightarrow\ n\#(m\times n) \ =\ n\#m + 1$

(a) Make a 3×3 chart of the values of $(0,..3)\#(0,..3)$.

(b) Show that the axioms become inconsistent if the antecedent of the second axiom is removed.

(c) How should we change the axioms to allow # to have extended natural operands?

41 For naturals n and m, we can express the statement "n is a factor of m" formally as follows:

$m: n \times nat$

(a) What are the factors of 0?
(b) What is 0 a factor of?
(c) What are the factors of 1?
(d) What is 1 a factor of?

42 Let $B = 1, 3, 5$. What is
(a) $¢(B + B)$
(b) $¢(B \times 2)$
(c) $¢(B \times B)$
(d) $¢(B^2)$

43 The compound axiom says

$x: A, B = x: A \vee x: B$

There are 16 two-operand boolean operators that could sit where $\vee$ sits in this axiom if we just replace bunch union (,) by a corresponding bunch operator. Which of the 16 two-operand boolean operators correspond to useful bunch operators?

44 (von Neumann numbers)
(a) Is there any harm in adding the axioms

$0 = \{null\}$	the empty set
$n+1 = \{n, \sim n\}$	for each natural n

(b) What correspondence is induced by these axioms between the arithmetic operations and the set operations?
(c) Is there any harm in adding the axioms

$0 = \{null\}$	the empty set
$i+1 = \{i, \sim i\}$	for each integer i

45 (Cantor's paradise) Show that $¢ł S > ¢S$ is neither a theorem nor an antitheorem.

46 The strings defined in Section 2.2 have natural indexes and extended natural lengths. Add a new operator, the inverse of catenation, to obtain strings that have negative indexes and lengths.

47 Prove the trichotomy for strings of numbers. For strings S and T, prove that exactly one of $S<T$, $S=T$, $S>T$ is a theorem.

48 In Section 2.3 there is a self-describing expression. Make it into a self-printing program. To do so, you need to know that $c!e$ outputs the value of expression e on channel c.

49 Simplify (no proof)
(a) $null, nil$
(b) $null; nil$
(c) $*nil$
(d) $[null]$
(e) $[*null]$

50 What is the difference between $[0, 1, 2]$ and $[0; 1; 2]$?

51 (prefix order) Give axioms to define the prefix partial order on strings. String S comes before string T in this order if and only if S is an initial segment of T .

52 Simplify, assuming i: $0,..\#L$
(a) $i \rightarrow Li \mid L$
(b) $L\,[0;..i] + [x] + L\,[i+1;..\#L]$

53 Simplify (no proof)
(a) $0 \rightarrow 1 \mid 1 \rightarrow 2 \mid 2 \rightarrow 3 \mid 3 \rightarrow 4 \mid 4 \rightarrow 5 \mid [0;..5]$
(b) $(4 \rightarrow 2 \mid [-3;..3])\ 3$
(c) $((3;2) \rightarrow [10;..15] \mid 3 \rightarrow [5;..10] \mid [0;..5])\ 3$
(d) $([0;..5]\ [3; 4])\ 1$
(e) $(2;2) \rightarrow$ "j" $\mid$ [["abc"]; ["de"]; ["fghi"]]
(f) $\#[nat]$
(g) $\#[*3]$
(h) $[3; 4]$: $[3*4*int]$
(i) $[3; 4]$: $[3; int]$
(j) $[3, 4; 5]$: $[2*int]$
(k) $[(3, 4); 5]$: $[2*int]$
(l) $[3; (4, 5); 6; (7, 8, 9)]$ ‘ $[3; 4; (5, 6); (7, 8)]$

54 Let i and j be indexes of list L . Express $i \rightarrow Lj \mid j \rightarrow Li \mid L$ without using $\mid$.

End of Basic Data Structures

10.3 Function Theory

55 In each of the following, replace p by
$\langle x: int \rightarrow \langle y: int \rightarrow \langle z: int \rightarrow x \geq 0 \wedge x^2 \leq y \wedge \forall z: int \cdot z^2 \leq y \Rightarrow z \leq x \rangle \rangle \rangle$
and simplify, assuming x, y, z, u, w: int .
(a) $p\ (x+y)\ (2 \times u + w)\ z$
(b) $p\ (x+y)\ (2 \times u + w)$
(c) $p\ (x+z)\ (y+y)\ (2+z)$

56 Some mathematicians like to use a notation like $\exists! x: D \cdot Px$ to mean “there is a unique x in D such that Px holds”. Define $\exists!$ formally.

57 Write an expression equivalent to each of the following without using $\S$.
(a) $\not\in(\S x: D \cdot Px) = 0$
(b) $\not\in(\S x: D \cdot Px) = 1$
(c) $\not\in(\S x: D \cdot Px) = 2$

58 (cat) Define function cat so that it applies to a list of lists and produces their catenation. For example,
$cat\ [[0; 1; 2]; [nil]; [[3]]; [4; 5]] = [0; 1; 2; [3]; 4; 5]$

59 Express formally that L is a sublist (not necessarily consecutive items) of list M . For example, $[0; 2; 1]$ is a sublist of $[0; 1; 2; 2; 1; 0]$, but $[0; 2; 1]$ is not a sublist of $[0; 1; 2; 3]$.

60 Express formally that L is a longest sorted sublist of M where
(a) the sublist must be consecutive items (a segment).
(b) the sublist must be consecutive (a segment) and nonempty.
(c) the sublist contains items in their order of appearance in M, but not necessarily consecutively (not necessarily a segment).

61 Express formally that natural n is the length of a longest palindromic segment in list L. A palindrome is a list that equals its reverse.

62 Using the syntax x **can fool** y **at time** t formalize the statements
(a) You can fool some of the people all of the time.
(b) You can fool all of the people some of the time.
(c) You can't fool all of the people all of the time.
for each of the following interpretations of the word "You":
(i) Someone
(ii) Anyone
(iii) The person I am talking to

63 (whodunit) Here are ten statements.
(i) Some criminal robbed the Russell mansion.
(ii) Whoever robbed the Russell mansion either had an accomplice among the servants or had to break in.
(iii) To break in one would have to either smash the door or pick the lock.
(iv) Only an expert locksmith could pick the lock.
(v) Anyone smashing the door would have been heard.
(vi) Nobody was heard.
(vii) No one could rob the Russell mansion without fooling the guard.
(viii) To fool the guard one must be a convincing actor.
(ix) No criminal could be both an expert locksmith and a convincing actor.
(x) Some criminal had an accomplice among the servants.
(a) Choosing good abbreviations, translate each of these statements into formal logic.
(b) Taking the first nine statements as axioms, prove the tenth.

64 (arity) The arity of a function is the number of variables (parameters) it introduces, and the number of arguments it can be applied to. Write axioms to define αf (arity of f).

65 There are some people, some keys, and some doors. Let p *holds* k mean that person p holds key k. Let k *unlocks* d mean that key k unlocks door d. Let p *opens* d mean that person p can open door d. Formalize
(a) Anyone can open any door if they have the appropriate key.
(b) At least one door can be opened without a key (by anyone).
(c) The locksmith can open any door even without a key.

66 Prove that if variables i and j do not appear in predicates P and Q, then
$$(\forall i\cdot Pi) \Rightarrow (\exists i\cdot Qi) \;=\; (\exists i, j\cdot Pi \Rightarrow Qj)$$

67 There are four boolean two-operand associative symmetric operators with an identity. We used two of them to define quantifiers. What happened to the other two?

68 Which operator can be used to define a quantifier to give the range of a function?

69 We have defined several quantifiers by starting with an associative symmetric operator with an identity. Bunch union is also such an operator. Does it yield a quantifier?

70 Exercise 13 talks about drinking and driving, but not about time. It's not all right to drink first and then drive soon after, but it is all right to drive first and then drink soon after. It is also all right to drink first and then drive 6 hours after. Let *drink* and *drive* be predicates of time, and formalize the rule that you can't drive for 6 hours after drinking. What does your rule say about drinking and driving at the same time?

71 Formalize each of the following statements as a boolean expression.
(a) Everybody loves somebody sometime.
(b) Every 10 minutes someone in New York City gets mugged.
(c) Every 10 minutes someone keeps trying to reach you.
(d) Whenever the altititude is below 1000 feet, the landing gear must be down.
(e) I'll see you on Tuesday, if not before.
(f) No news is good news.

72 Express formally that
(a) natural n is the largest proper (neither 1 nor m) factor of natural m.
(b) g is the greatest common divisor of naturals a and b.
(c) m is the lowest common multiple of naturals a and b.
(d) p is a prime number.
(e) n and m are relatively prime numbers.
(f) there is at least one and at most a finite number of naturals satisfying predicate p.
(g) there is no smallest integer.
(h) between every two rational numbers there is another rational number.
(i) list L is a longest segment of list M that does not contain item x.
(j) the segment of list L from (including) index i to (excluding) index j is a segment whose sum is smallest.
(k) a and b are items of lists A and B (respectively) whose absolute difference is least.
(l) p is the length of a longest plateau (segment of equal items) in a nonempty sorted list L.
(m) all items that occur in list L occur in a segment of length 10.
(n) all items of list L are different (no two items are equal).
(o) at most one item in list L occurs more than once.
(p) the maximum item in list L occurs m times.
(q) list L is a permutation of list M.

73 (bitonic list) A list is bitonic if it is monotonic up to some index, and antimonotonic after that. For example, $[1; 3; 4; 5; 5; 6; 4; 4; 3]$ is bitonic. Express formally that L is bitonic.

74 Formalize and disprove the statement “There is a natural number that is not equal to any natural number.”.

75 (friends) Formalize and prove the statement “The people you know are those known by all who know all whom you know.”.

76 (swapping partners) There is a finite bunch of couples. Each couple consists of a man and a woman. The oldest man and the oldest woman have the same age. If any two couples swap partners, forming two new couples, the younger partners of the two new couples have the same age. Prove that in each couple, the partners have the same age.

77 Express $\forall$ and $\exists$ in terms of $\not\in$ and $\S$.

78 Simplify
(a) $\Sigma\ ((0,..n) \rightarrow m)$
(b) $\Pi\ ((0,..n) \rightarrow m)$
(c) $\forall\ ((0,..n) \rightarrow b)$
(d) $\exists\ ((0,..n) \rightarrow b)$

79 Are the boolean expressions
$nil \rightarrow x\ =\ x$
$(S;T) \rightarrow x\ =\ S \rightarrow T \rightarrow x$
(a) consistent with the theory in Chapters 2 and 3?
(b) theorems according to the theory in Chapters 2 and 3?

80 (unicorns) The following statements are made.
All unicorns are white.
All unicorns are black.
No unicorn is both white and black.
Are these statements consistent? What, if anything, can we conclude about unicorns?

81 (Russell's barber) Bertrand Russell stated: "In a small town there is a barber who shaves all and only the people in the town who do not shave themselves.". Then Russell asked: "Does the barber shave himself?". If we say yes, then we can conclude from the statement that he does not, and if we say no, then we can conclude from the statement that he does. Formalize this paradox, and thus explain it.

82 (Russell's paradox) Define $rus\ =\ \langle f\colon (null \rightarrow bool) \rightarrow \neg\ f\ f \rangle$.
(a) Can we prove $rus\ rus\ =\ \neg\ rus\ rus$?
(b) Is this an inconsistency?
(c) Can we add the axiom $\neg\ f\colon \Delta f$? Would it help?

83 Prove that the square of an odd natural number is odd, and the square of an even natural number is even.

84 (Gödel/Turing incompleteness) Prove that we cannot consistently and completely define an interpreter. An interpreter is a predicate $\mathbb{I}$ that applies to texts; when applied to a text representing a boolean expression, its result is equal to the represented expression. For example,
$\mathbb{I}\ \text{"}\forall s\colon [*char]\cdot\ \#s \geq 0\text{"}\ =\ \forall s\colon [*char]\cdot\ \#s \geq 0$

85 Let f and g be functions from nat to nat . For what f do we have the theorem $g\,f = g$? For what f do we have the theorem $f\,g = g$?

86 What is the difference between $\#[n*\mathbb{T}]$ and $\not\in\S[n*\mathbb{T}]$?

87 Without using the Bounding Laws, prove
(a) $\forall i\cdot\ Li \leq m\ =\ (MAX\ L) \leq m$
(b) $\exists i\cdot\ Li \leq m\ =\ (MIN\ L) \leq m$

88 (pigeon-hole) Prove $(\Sigma L) > n \times \#L\ \Rightarrow\ \exists i\colon \Delta L\cdot\ Li > n$.

89 If $f: A \to B$ and $p: B \to bool$, prove

(a) $\exists b: fA \cdot pb = \exists a: A \cdot pfa$

(b) $\forall b: fA \cdot pb = \forall a: A \cdot pfa$

90 This question explores a simpler, more elegant function theory than the one presented in Chapter 3. We separate the notion of local variable introduction from the notion of domain, and we generalize the latter to become local axiom introduction. Variable introduction has the form $\langle v \to b \rangle$ where v is a variable and b is any expression (the body; no domain). There is an Application Law

$\langle v \to b \rangle\, x =$ (substitute x for v in b)

and an Extension Law

$f = \langle v \to fv \rangle$

Let a be boolean, and let b be any expression. Then $a \mathbin{»} b$ is an expression of the same type as b. The $»$ operator has precedence level 12 and is right-associating. Its axioms include:

$\top \mathbin{»} b = b$

$a \mathbin{»} b \mathbin{»} c = a \wedge b \mathbin{»} c$

The expression $a \mathbin{»} b$ is a "one-tailed if-expression", or "asserted expression"; it introduces a as a local axiom within b. A function is a variable introduction whose body is an asserted expression in which the assertion has the form $v: D$. In this case, we allow an abbreviation: for example, the function $\langle n \to n: nat \mathbin{»} n+1 \rangle$ can be abbreviated $\langle n: nat \to n+1 \rangle$. Applying this function to 3, we find

$\langle n \to n: nat \mathbin{»} n+1 \rangle\, 3$

$= 3: nat \mathbin{»} 3+1$

$= \top \mathbin{»} 4$

$= 4$

Applying it to -3 we find

$\langle n \to n: nat \mathbin{»} n+1 \rangle\, (-3)$

$= -3: nat \mathbin{»} -3+1$

$= \bot \mathbin{»} -2$

and then we are stuck; no further axiom applies. In the example, we have used variable introduction and axiom introduction together to give us back the kind of function we had; but in general, they are independently useful.

(a) Show how function-valued variables can be introduced in this new theory.

(b) What expressions in the old theory have no equivalent in the new? How closely can they be approximated?

(c) What expressions in the new theory have no equivalent in the old? How closely can they be approximated?

91 Is there any harm in defining relation R with the following axioms?

$\forall x \cdot \exists y \cdot Rxy$	totality
$\forall x \cdot \neg Rxx$	irreflexivity
$\forall x, y, z \cdot Rxy \wedge Ryz \Rightarrow Rxz$	transitivity
$\exists u \cdot \forall x \cdot x=u \vee Rxu$	unity

92 Let n be a natural number, and let R be a relation on $0,..n$. In other words,

$R: (0,..n) \to (0,..n) \to bool$

We say that from x we can reach x in zero steps. If Rxy we say that from x we can reach y in one step. If Rxy and Ryz we say that from x we can reach z in two steps. And so on. Express formally that from x we can reach y in some number of steps.

93 Relation R is transitive if $\forall x, y, z \cdot Rxy \wedge Ryz \Rightarrow Rxz$. Express formally that relation R is the transitive closure of relation Q (R is the strongest transitive relation that is implied by Q).

End of Function Theory

10.4 Program Theory

94 Prove specification S is satisfiable for prestate σ if and only if $S.\mathsf{T}$ (note: T is the "true" boolean).

95 Let x be an integer state variable. Which of the following specifications are implementable?

(a) $x \geq 0 \Rightarrow x'^2 = x$

(b) $x' \geq 0 \Rightarrow x = 0$

(c) $\neg(x \geq 0 \wedge x' = 0)$

(d) $\neg(x \geq 0 \vee x' = 0)$

96 A specification is transitive if, for all states a, b, and c, if it allows the state to change from a to b, and it allows the state to change from b to c, then it allows the state to change from a to c. Prove S is transitive if and only if S is refined by $S.S$.

97√ Simplify each of the following (in integer variables x and y).

(a) $x := y+1.\ y'>x'$

(b) $x := x+1.\ y'>x \wedge x'>x$

(c) $x := y+1.\ y'=2x$

(d) $x := 1.\ x \geq 1 \Rightarrow \exists x \cdot y'=2x$

(e) $x := y.\ x \geq 1 \Rightarrow \exists y \cdot y'=x \times y$

(f) $x := 1.\ ok$

(g) $x := 1.\ y := 2$

(h) $x := 1.\ P$ where $P = y := 2$

(i) $x := 1.\ y := 2.\ x := x+y$

(j) $x := 1.$ **if** $y>x$ **then** $x := x+1$ **else** $x := y$

(k) $x := 1.\ x'>x.\ x'=x+1$

98 Prove

(a) $x := x = ok$

(b) $x := e.\ x := f\,x = x := f\,e$

99 Prove or disprove

(a) $R.$ **if** b **then** P **else** $Q = $ **if** b **then** $(R.P)$ **else** $(R.Q)$

(b) **if** b **then** $P \Rightarrow Q$ **else** $R \Rightarrow S = $ (**if** b **then** P **else** R) $\Rightarrow$ (**if** b **then** Q **else** S)

(c) **if** b **then** $(P.Q)$ **else** $(R.S) = $ **if** b **then** P **else** $R.$ **if** b **then** Q **else** S

100 Prove

(a) P and Q are each refined by R if and only if their conjunction is refined by R.

(b) $P \Rightarrow Q$ is refined by R if and only if Q is refined by $P \wedge R$.

101 (rolling)

(a) Can we always unroll a loop? If $S \Leftarrow A.S.Z$, can we conclude $S \Leftarrow A.A.S.Z.Z$?

(b) Can we always roll up a loop? If $S \Leftarrow A.A.S.Z.Z$, can we conclude $S \Leftarrow A.S.Z$?

102 What is wrong with the following proof:

$$\begin{array}{lll} & (R \Leftarrow R.S) & \text{use context rule} \\ = & (R \Leftarrow \bot.S) & \bot \text{ is base for } . \\ = & (R \Leftarrow \bot) & \text{base law for } \Leftarrow \\ = & \mathsf{T} & \end{array}$$

103 For which kinds of specifications P and Q is the following a theorem:

(a) $\neg(P.\neg Q) \Leftarrow P.Q$

(b) $P.Q \Leftarrow \neg(P.\neg Q)$

(c) $P.Q = \neg(P.\neg Q)$

104 Write a formal specification of the following problem: "Change the value of list variable L so that each item is repeated. For example, if L is [6; 3; 5; 5; 7] then it should be changed to [6; 6; 3; 3; 5; 5; 5; 5; 7; 7] .".

105 Let P and Q be specifications. Let C be a precondition and let C' be the corresponding postcondition. Prove the condition law

$$P.Q \Leftarrow P \wedge C'.\ C \Rightarrow Q$$

106 Let P and Q be specifications. Let C be a precondition and let C' be the corresponding postcondition. Which three of the following condition laws can be turned around, switching the problem and the solution?

$C \wedge (P.Q) \Leftarrow C \wedge P.Q$
$C \Rightarrow (P.Q) \Leftarrow C \Rightarrow P.Q$
$(P.Q) \wedge C' \Leftarrow P.Q \wedge C'$
$(P.Q) \Leftarrow C' \Leftarrow P.Q \Leftarrow C'$
$P.C \wedge Q \Leftarrow P \wedge C'.Q$
$P.Q \Leftarrow P \wedge C'.\ C \Rightarrow Q$

107 Let S be a specification. Let C be a precondition and let C' be the corresponding postcondition. How does the exact precondition for C' to be refined by S differ from $(S.C)$? Hint: consider prestates in which S is unsatisfiable, then deterministic, then nondeterministic.

108 We have Refinement by Steps, Refinement by Parts, and Refinement by Cases. In this question we propose Refinement by Alternatives:
If $A \Leftarrow$ **if** b **then** C **else** D and $E \Leftarrow$ **if** b **then** F **else** G are theorems,
then $A \vee E \Leftarrow$ **if** b **then** $C \vee F$ **else** $D \vee G$ is a theorem.
If $A \Leftarrow B.C$ and $D \Leftarrow E.F$ are theorems, then $A \vee D \Leftarrow B \vee E.\ C \vee F$ is a theorem.
If $A \Leftarrow B$ and $C \Leftarrow D$ are theorems, then $A \vee C \Leftarrow B \vee D$ is a theorem.
Discuss the merits and demerits of this proposed law.

109 Let x and y be real variables. Prove that if $y = x^2$ is true before
$x := x+1.\ y := y + 2 \times x - 1$
is executed, then it is still true after.

110√ In one integer variable x ,

(a) find the exact precondition A for $x' > 5$ to be refined by $x := x+1$.

(b) find the exact postcondition for A to be refined by $x := x+1$, where A is your answer from part (a).

111 Let all variables be integer except L is a list of integers. What is the exact precondition for

(a) $x'+y' > 8$ to be refined by $x := 1$

(b) $x'=1$ to be refined by $x := 1$

(c) $x'=2$ to be refined by $x := 1$

(d) $x'=y$ to be refined by $y := 1$

(e) $x' \geq y'$ to be refined by $x := y+z$

(f) $y'+z' \geq 0$ to be refined by $x := y+z$

(g) $x' \leq 1 \ \vee \ x' \geq 5$ to be refined by $x := x+1$

(h) $x' < y' \ \wedge \ \exists x \cdot Lx < y'$ to be refined by $x := 1$

(i) $\exists y \cdot Ly < x'$ to be refined by $x := y+1$

(j) $L'\ 3 = 4$ to be refined by $L := i \rightarrow 4 \mid L$

(k) $x'=a$ to be refined by **if** $a > b$ **then** $x := a$ **else** *ok*

(l) $x'=y \ \wedge \ y'=x$ to be refined by $(z := x.\ x := y.\ y := z)$

(m) $a \times x'^{\,2} + b \times x' + c = 0$ to be refined by $(x := a \times x + b.\ x := -x/a)$

(n) $f' = n'!$ to be refined by $(n := n+1.\ f := f \times n)$ where n is natural and $!$ is factorial.

(o) $7 \leq c' < 28 \ \wedge \ odd\ c'$ to be refined by $(a := b-1.\ b := a+3.\ c := a+b)$

(p) $s' = \Sigma\, L\,[0;..i']$ to be refined by $(s := s + Li.\ i := i+1)$

112 For what exact precondition and postcondition does the following assignment move integer variable x farther from zero?

(a) $x := x+1$

(b) $x := abs\ (x+1)$

(c)√ $x := x^2$

113 For what exact precondition and postcondition does the following assignment move integer variable x farther from zero staying on the same side of zero?

(a) $x := x+1$

(b) $x := abs\ (x+1)$

(c) $x := x^2$

114 Prove

(a) the Precondition Law: C is a sufficient precondition for specification P to be refined by specification S if and only if $C \Rightarrow P$ is refined by S.

(b) the Postcondition Law: C' is a sufficient postcondition for specification P to be refined by specification S if and only if $C' \Rightarrow P$ is refined by S.

115 (weakest prespecification, weakest postspecification) Given specifications P and Q, find the weakest specification S (in terms of P and Q) such that P is refined by

(a) $S.\ Q$

(b) $Q.\ S$

116 Let a, b, and c be integer variables. Simplify

(a) $b := a-b.\ b := a-b$

(b) $a := a+b.\ b := a-b.\ a := a-b$

(c) $c := a-b-c.\ b := a-b-c.\ a := a-b-c.\ c := a+b+c$

117 Let x and y be boolean variables. Simplify

(a) $x := x=y.\ x := x=y$

(b) $x := x \neq y.\ y := x \neq y.\ x := x \neq y$

118 Let x be an integer variable. Prove the refinement

(a)√ $x'=0 \Leftarrow$ **if** $x=0$ **then** ok **else** $(x:= x-1.\ x'=0)$

(b) $P \Leftarrow$ **if** $x=0$ **then** ok **else** $(x:= x-1.\ t:= t+1.\ P)$

where $P = x'=0 \wedge$ **if** $x\geq 0$ **then** $t' = t+x$ **else** $t'=\infty$

119 Let x be an integer variable. Prove the refinement

(a) $x'=1 \Leftarrow$ **if** $x=1$ **then** ok **else** $(x:= div\ x\ 2.\ x'=1)$

(b)√ $R \Leftarrow$ **if** $x=1$ **then** ok **else** $(x:= div\ x\ 2.\ t:= t+1.\ R)$

where $R = x'=1 \wedge$ **if** $x\geq 1$ **then** $t' \leq t + log\ x$ **else** $t'=\infty$

120 In natural variables s and n prove

$P \Leftarrow$ **if** $n=0$ **then** ok **else** $(n:= n-1.\ s:= s+2^n-n.\ t:= t+1.\ P)$

where $P = s' = s + 2^n - n\times(n-1)/2 - 1 \wedge n'=0 \wedge t' = t+n$.

121 Let x be an integer variable. Is the refinement

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

a theorem when

$P = x<0 \Rightarrow x'=1 \wedge t'=\infty$

Is this reasonable? Explain.

122 (factorial) In natural variables n and f prove

$f:= n! \Leftarrow$ **if** $n=0$ **then** $f:= 1$ **else** $(n:= n-1.\ f:= n!.\ n:= n+1.\ f:= f\times n)$

where $n! = 1\times 2\times 3\times ...\times n$.

123 In natural variables n and m prove

$P \Leftarrow$ $n:= n+1.$
if $n=10$ **then** ok
else $(m:= m-1.\ P)$

where $P = m:= m+n-9.\ n:= 10$.

124 Let x and n be natural variables. Find a specification P such that both the following hold:

$x = x'\times 2^{n'} \Leftarrow n:= 0.\ P$

$P \Leftarrow$ **if** $even\ x$ **then** $(x:= x/2.\ n:= n+1.\ P)$ **else** ok

125 (square) Let s and n be natural variables. Find a specification P such that both the following hold:

$s' = n^2 \Leftarrow s:= n.\ P$

$P \Leftarrow$ **if** $n=0$ **then** ok **else** $(n:= n-1.\ s:= s+n+n.\ P)$

This program squares using only addition, subtraction, and test for zero.

126 Let a and b be positive integers. Let x , u , and v be integer variables. Let

$P = u\geq 0 \wedge v\geq 0 \wedge x = u\times a - v\times b \Rightarrow x'=0$

(a) Prove

$P \Leftarrow$ **if** $x>0$ **then** $(x:= x-a.\ u:= u-1.\ P)$
else if $x<0$ **then** $(x:= x+b.\ v:= v-1.\ P)$
else ok

(b) Find an upper bound for the execution time of the program in part (a).

127 Let i be an integer variable. Add time according to the recursive measure, and then find the strongest P you can such that

(a) $P \Leftarrow$ **if** *even* i **then** $i := i/2$ **else** $i := i+1$.
if $i=1$ **then** *ok* **else** P

(b) $P \Leftarrow$ **if** *even* i **then** $i := i/2$ **else** $i := i-3$.
if $i=0$ **then** *ok* **else** P

128 Find a finite function f of natural variables i and j to serve as an upper bound on the execution time of the following program, and prove

$t' \le t + fij \Leftarrow$ **if** $i=0 \wedge j=0$ **then** *ok*
else if $i=0$ **then** ($i := j \times j$. $j := j-1$. $t := t+1$. $t' \le t + fij$)
else ($i := i-1$. $t := t+1$. $t' \le t + fij$)

129 Let P mean that the final values of natural variables a and b are the largest exponents of 2 and 3 respectively such that both powers divide evenly into the initial value of positive integer x.

(a) Define P formally.

(b) Define Q suitably and prove

$P \Leftarrow a := 0$. $b := 0$. Q
$Q \Leftarrow$ **if** x: $2 \times nat$ **then** ($x := x/2$. $a := a+1$. Q)
else if x: $3 \times nat$ **then** ($x := x/3$. $b := b+1$. Q)
else *ok*

(c) Find an upper bound for the execution time of the program in part (b).

130 Express formally that specification R is satisfied by any number (including 0) of repetitions of behavior satisfying specification S.

131 (Zeno) Here is a loop.

$R \Leftarrow x := x+1$. R

Suppose we charge time 2^{-x} for the recursive call, so that each iteration takes half as long as the one before. Prove that the execution time is finite.

132 Let t be the time variable. Can we prove the refinement

$P \Leftarrow t := t+1$. P

for $P = t'=5$? Does this mean that execution will terminate at time 5 ? What is wrong?

133 Let n and r be natural variables in the refinement

$P \Leftarrow$ **if** $n=1$ **then** $r := 0$ **else** ($n :=$ *div* n 2. P. $r := r+1$)

Suppose the operations *div* and + each take time 1 and all else is free (even the call is free). Insert appropriate time increments, and find an appropriate P to express the execution time in terms of

(a) the initial values of the memory variables. Prove the refinement for your choice of P.

(b) the final values of the memory variables. Prove the refinement for your choice of P.

134 (running total) Given list variable L and any other variables you need, write a program to convert L into a list of cumulative sums. Formally,

(a) $\forall n$: 0,..$\#L \cdot L'n = \Sigma L\,[0;..n]$

(b) $\forall n$: 0,..$\#L \cdot L'n = \Sigma L\,[0;..n+1]$

135 (cube) Write a program that cubes using only addition, subtraction, and test for zero.

136 (cube test) Write a program to determine if a given natural number is a cube without using exponentiation.

137 (*mod* 2) Let n be a natural variable. The problem to reduce n modulo 2 can be solved as follows:

$n' = mod\ n\ 2 \Leftarrow$ **if** $n<2$ **then** *ok* **else** (n:= $n–2$. $n' = mod\ n\ 2$)

Using the recursive time measure, find and prove an upper time bound. Make it as small as you can.

138 (fast *mod* 2) Let n and p be natural variables. The problem to reduce n modulo 2 can be solved as follows:

$n' = mod\ n\ 2 \Leftarrow$ **if** $n<2$ **then** *ok* **else** ($even\ n' = even\ n$. $n' = mod\ n\ 2$)

$even\ n' = even\ n \Leftarrow p := 2.\ even\ p \Rightarrow even\ p' \wedge even\ n' = even\ n$

$even\ p \Rightarrow even\ p' \wedge even\ n' = even\ n \Leftarrow$

$n := n–p.\ p := p+p.$

if $n<p$ **then** *ok* **else** $even\ p \Rightarrow even\ p' \wedge even\ n' = even\ n$

(a) Prove these refinements.

(b) Using the recursive time measure, find and prove a sublinear upper time bound.

139 Given a specification P and a prestate σ with t as time variable, we might define "the exact precondition for termination" as follows:

$\exists n: nat \cdot \forall \sigma' \cdot\ t' \le t+n \Leftarrow P$

Letting x be an integer variable, find the exact precondition for termination of the following, and comment on whether it is reasonable.

(a) $x \ge 0 \Rightarrow t' \le t+x$

(b) $\exists n: nat \cdot\ t' \le t+n$

(c) $\exists f: int \to nat \cdot\ t' \le t + fx$

140√ (maximum item) Write a program to find the maximum item in a list.

141 (list comparison) Using item comparison but not list comparison, write a program to determine whether one list comes before another in the list order.

142√ (list summation) Write a program to find the sum of a list of numbers.

143 (alternating sum) Write a program to find the alternating sum $L0 – L1 + L2 – L3 + \ldots$ of a list L of numbers.

144 (combinations) Write a program to find the number of ways to partition $a+b$ things into a things in the left part and b things in the right part. Include recursive time.

145 (earliest meeting time) Write a program to find the earliest meeting time acceptable to three people. Each person is willing to state their possible meeting times by means of a function that tells, for each time t, the earliest time at or after t that they are available for a meeting. (Do not confuse this t with the execution time variable. You may ignore execution time for this problem.)

146 (polynomial) You are given n: *nat* , c: [n**rat*] , x: *rat* and variable y: *rat* . c is a list of coefficients of a polynomial ("of degree n–1") to be evaluated at x . Write a program for

$y' = \Sigma i: 0,..n \cdot\ ci \times x^i$

147 (multiplication table) Given n: *nat* and variable M: [*[**nat*]] , write a program to assign to M a multiplication table of size n without using multiplication. For example, if $n = 4$, then

M' = [[0];
[0; 1];
[0; 2; 4];
[0; 3; 6; 9]]

148 (Pascal's triangle) Given n: *nat* and variable P: [*[**nat*]] , write a program to assign to P a Pascal's triangle of size n . For example, if $n = 4$, then

P' = [[1];
[1; 1];
[1; 2; 1];
[1; 3; 3; 1]]

The left side and diagonal are all 1s; each interior item is the sum of the item above it and the item diagonally above and left.

149√ (binary exponentiation) Given natural variables x and y , write a program for $y' = 2^x$ without using exponentiation.

150 Write a program to find the smallest power of 2 that is bigger than or equal to a given positive integer without using exponentiation.

151√ (fast exponentiation) Given rational variables x and z and natural variable y , write a program for $z' = x^y$ that runs fast without using exponentiation.

152 (sort test) Write a program to assign a boolean variable to indicate whether a given list is sorted.

153√ (linear search) Write a program to find the first occurrence of a given item in a given list. The execution time must be linear in the length of the list.

154√ (binary search) Write a program to find a given item in a given nonempty sorted list. The execution time must be logarithmic in the length of the list. The strategy is to identify which half of the list contains the item if it occurs at all, then which quarter, then which eighth, and so on.

155 (binary search with test for equality) The problem is binary search (Exercise 154), but each iteration tests to see if the item in the middle of the remaining segment is the item we seek.

(a) Write the program, with specifications and proofs.
(b) Find the execution time according to the recursive measure.
(c) Find the execution time according to a measure that charges time 1 for each test.
(d) Compare the execution time to binary search without the test for equality each iteration.

156 (ternary search) The problem is the same as binary search (Exercise 154). The strategy this time is to identify which third of the list contains the item if it occurs at all, then which ninth, then which twenty-seventh, and so on.

157√ (two-dimensional search) 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.

158 (sorted two-dimensional search) Write a program to find a given item in a given 2-dimensional array in which each row is sorted and each column is sorted. The execution time must be linear in the sum of the dimensions.

159 (sorted two-dimensional count) Write a program to count the number of occurrences of a given item in a given 2-dimensional array in which each row is sorted and each column is sorted. The execution time must be linear in the sum of the dimensions.

160 (pattern search) Let *subject* and *pattern* be two texts. Write a program to do the following. If *pattern* occurs somewhere within *subject*, natural variable h is assigned to indicate the beginning of its first occurrence

(a) using any list operators given in Section 2.3.

(b) using list indexing, but no other list operators.

161 (fixed point) Let L be a nonempty sorted list of n different integers. Write a program to find a fixed-point of L, that is an index i such that $Li = i$, or to report that no such index exists. Execution time should be at most $log\ n$ where n is the length of the list.

162 (all present) Given a natural number and a list, write a program to determine if every natural number up to the given number is an item in the list.

163 (missing number) You are given an unsorted list of length n whose items are the numbers $0,..n+1$ with one number missing. Write a program to find the missing number.

164 (text length) You are given a text (string of characters) that begins with zero or more "ordinary" characters, and then ends with zero or more "padding" characters. A padding character is not an ordinary character. Write a program to find the number of ordinary characters in the text. Execution time should be logarithmic in the text length.

165 (ordered pair search) Given a list of at least two items whose first item is less than or equal to its last item, write a program to find an adjacent pair of items such that the first of the pair is less than or equal to the second of the pair. Execution time should be logarithmic in the length of the list.

166 (convex equal pair) A list of numbers is convex if its length is at least 2, and every item (except the first and last) is less than or equal to the average of its two neighbors. Given a convex list, write a program to determine if it has a pair of consecutive equal items. Execution should be logarithmic in the length of the list.

167 Define a partial order $\ll$ on pairs of integers as follows:

$$[a; b] \ll [c; d] \quad = \quad a<c \wedge b<d$$

Given n: $nat+1$ and L: $[n*[int; int]]$ write a program to find the index of a minimal item in L. That is, find j: $0,..\#L$ such that $\neg\exists i\cdot Li \ll Lj$. The execution time should be at most $n \times log\ n$.

168 (n sort) Given a list L such that $L(0,..\#L) = 0,..\#L$, write a program to sort L in linear time and constant space. The only change permitted to L is to swap two items.

169√ (n^2 sort) Write a program to sort a list. Execution time should be at most n^2 where n is the length of the list.

170 ($n \times log\ n$ sort) Write a program to sort a list. Execution time should be at most $n \times log\ n$ where n is the length of the list.

171 (reverse) Write a program to reverse the order of the items of a list.

172 (next sorted list) Given a nonempty sorted list of naturals, write a program to find the next (in list order) sorted list having the same length and sum.

173 (next combination) You are given a sorted list of m different numbers, all in the range $0,..n$. Write a program to find the lexicographically next sorted list of m different numbers, all in the range $0,..n$.

174 (next permutation) You are given a list of the numbers $0,..n$ in some order. Write a program to find the lexicographically next list of the numbers $0,..n$.

175 (permutation inverse) You are given a list variable P of different items in $0,..\#P$. Write a program for $P\ P' = [0;..\#P]$.

176 (idempotent permutation) You are given a list variable L of items in $0,..\#L$ (not necessarily all different). Write a program to permute the list so that finally $L'\ L' = L'$.

177 (local minimum) You are given a list L of at least 3 numbers such that $L0 \geq L1$ and $L(\#L–2) \leq L(\#L–1)$. A local minimum is an interior index i: $1,..\#L–1$ such that

$$L(i–1) \geq Li \leq L(i+1)$$

Write a program to find a local minimum of L .

178 (natural division) The natural quotient of natural n and positive integer p is the natural number q satisfying

$$q \leq n/p < q+1$$

Write a program to find the natural quotient of n and p in $log\ n$ time without using any functions (*div* , *mod* , *floor* , *ceil* , ...).

179 (remainder) Write a program to find the remainder after natural division (Exercise 178), using only comparison, addition, and subtraction (not multiplication or division or *mod*).

180 (natural binary logarithm) The natural binary logarithm of a positive integer p is the natural number b satisfying

$$2^b \leq p < 2^{b+1}$$

Write a program to find the natural binary logarithm of a given positive integer p in $log\ p$ time.

181 (natural square root) The natural square root of a natural number n is the natural number s satisfying

$$s^2 \leq n < (s+1)^2$$

(a) Write a program to find the natural square root of a given natural number n in $log\ n$ time.

(b) Write a program to find the natural square root of a given natural number n in $log\ n$ time using only addition, subtraction, doubling, halving, and comparisons (no multiplication or division).

182 (factor count) Write a program to find the number of factors (not necessarily prime) of a given natural number.

183 (Fermat's last program) Given natural c, write a program to find the number of unordered pairs of naturals a and b such that $a^2 + b^2 = c^2$ in time proportional to c. (An unordered pair is really a bunch of size 1 or 2. If we have counted the pair a and b, we don't want to count the pair b and a.) Your program may use addition, subtraction, multiplication, division, and comparisons, but not exponentiation or square root.

184 (flatten) Write a program to flatten a list. The result is a new list just like the old one but without the internal structure. For example,

L = [[3; 5]; 2; [5; [7]; [*nil*]]]

L' = [3; 5; 2; 5; 7]

Your program may employ a test Li: *int* to see if an item is an integer or a list.

185 (diagonal) Some points are arranged around the perimeter of a circle. The distance from each point to the next point going clockwise around the perimeter is given by a list. Write a program to find two points that are farthest apart.

186 (minimum sum segment) Given a list of integers, possibly including negatives, write a program to find

(a)√ the minimum sum of any segment (sublist of consecutive items).

(b) the segment (sublist of consecutive items) whose sum is minimum.

187 (maximum product segment) Given a list of integers, possibly including negatives, write a program to find

(a) the maximum product of any segment (sublist of consecutive items).

(b) the segment (sublist of consecutive items) whose product is maximum.

188 (segment sum count)

(a) Write a program to find, in a given list of naturals, the number of segments whose sum is a given natural.

(b) Write a program to find, in a given list of positive naturals, the number of segments whose sum is a given natural.

189 (longest plateau) You are given a nonempty sorted list of numbers. A plateau is a segment (sublist of consecutive items) of equal items. Write a program to find

(a) the length of a longest plateau.

(b) the number of longest plateaus.

190 (longest smooth segment) In a list of integers, a smooth segment is a sublist of consecutive items in which no two adjacent items differ by more than 1. Write a program to find a longest smooth segment.

191 (longest balanced segment) Given a list of booleans, write a program to find a longest segment (sublist of consecutive items) having an equal number of $\top$ and $\bot$ items.

192 (longest palindrome) A palindrome is a list that equals its reverse. Write a program to find a longest palindromic segment in a given list.

193 (greatest subsequence) Given a list, write a program to find the sublist that is largest according to list order. (A sublist contains items drawn from the list, in the same order of appearance, but not necessarily consecutive items.)

194 Given a list whose items are all 0, 1, or 2, write a program

(a) to find the length of a shortest segment (consecutive items) that contains all three numbers in any order.

(b) to count the number of sublists (not necessarily consecutive items) that are 0 then 1 then 2 in that order.

195 Let L and M be sorted lists of numbers. Write a program to find the number of pairs of indexes $i: 0,..\#L$ and $j: 0,..\#M$ such that $Li \le Mj$.

196 (heads and tails) Let L be a list of positive integers. Write a program to find the number of pairs of indexes i and j such that

$$\Sigma\, L\,[0;..i] \;=\; \Sigma\, L\,[j;..\#L]$$

197 (pivot) You are given a nonempty list of positive numbers. Write a program to find the balance point, or pivot. Each item contributes its value (weight) times its distance from the pivot to its side of the balance. Item i is considered to be located at point $i + 1/2$, and the pivot point may likewise be noninteger.

198 (inversion count) Given a list, write a program to find how many pairs of items (not necessarily consecutive items) are out of order, with the larger item before the smaller item.

199 (minimum difference) Given two nonempty sorted lists of numbers, write a program to find a pair of items, one from each list, whose absolute difference is smallest.

200 (earliest quitter) In a nonempty list find the first item that is not repeated later. In list $[13; 14; 15; 14; 15; 13]$ the earliest quitter is 14 because the other items 13 and 15 both occur after the last occurrence of 14.

201 (interval union) A collection of intervals along a real number line is given by the list of left ends L and the corresponding list of right ends R. List L is sorted. The intervals might sometimes overlap, and sometimes leave gaps. Write a program to find the total length of the number line that is covered by these intervals.

202 (bit sum) Write a program to find the number of ones in the binary representation of a given natural number.

203 (digit sum) Write a program to find the sum of the digits in the decimal representation of a given natural number.

204 (parity check) Write a program to find whether the number of ones in the binary representation of a given natural number is even or odd.

205 (approximate search) Given a nonempty sorted list of numbers and a number, write a program to determine the index of an item in the list that is closest in value to the given number.

206 Given two natural numbers s and p, write a program to find four natural numbers a, b, c, and d whose sum is s and product p, in time s^2, if such numbers exist.

207 Given three natural numbers n, s, and p, write a program to find a list of length n of natural numbers whose sum is s and product p, if such a list exists.

208 (transitive closure) A relation R: $(0,..n)\rightarrow(0,..n)\rightarrow bool$ can be represented by a square boolean array of size n. Given a relation in the form of a square boolean array, write a program to find

(a) its transitive closure (the strongest transitive relation that is implied by the given relation).

(b) its reflexive transitive closure (the strongest reflexive and transitive relation that is implied by the given relation).

209 (reachability) You are given a finite bunch of places; and a successor function S on places that tells, for each place, those places that are directly reachable from it; and a special place named h (for home). Write a program to find all places that are reachable (reflexively, directly, or indirectly) from h.

210 (shortest path) You are given a square extended rational array in which item $i\,j$ represents the direct distance from place i to place j. If it is not possible to go directly from i to j, then item $i\,j$ is ∞. Write a program to find the square extended rational array in which item $i\,j$ represents the shortest, possibly indirect, distance from place i to place j.

211 (McCarthy's 91 problem) Let i be an integer variable. Let

$$M \;=\; \textbf{if } i{>}100 \textbf{ then } i{:=}\, i{-}10 \textbf{ else } i{:=}\, 91$$

(a) Prove $M \;\Leftarrow\; \textbf{if } i{>}100 \textbf{ then } i{:=}\, i{-}10 \textbf{ else } (i{:=}\, i{+}11.\ M.\ M)$.

(b) Find the execution time of M as refined in part (a).

212 (Towers of Hanoi) There are 3 towers and n disks. The disks are graduated in size; disk 0 is the smallest and disk $n{-}1$ is the largest. Initially tower A holds all n disks, with the largest disk on the bottom, proceding upwards in order of size to the smallest disk on top. The task is to move all the disks from tower A to tower B, but you can move only one disk at a time, and you must never put a larger disk on top of a smaller one. In the process, you can make use of tower C as intermediate storage.

(a)√ Using the command *MoveDisk from to* to cause a robot arm to move the top disk from tower *from* to tower *to*, write a program to move all the disks from tower A to tower B.

(b)√ Find the execution time, counting *MoveDisk* as time 1, and all else free.

(c) Suppose that the posts where the disks are placed are arranged in an equilateral triangle, so that the distance the arm moves each time is constant (one side of the triangle to get into position plus one side to move the disk), and not dependent on the disk being moved. Suppose the time to move a disk varies with the weight of the disk being moved, which varies with its area, which varies with the square of its radius, which varies with the disk number. Find the execution time.

(d)√ Find the maximum memory space required by the program, counting a recursive call as 1 location (for the return address) and all else free.

(e)√ Find the average memory space required by the program, counting a recursive call as 1 location (for the return address) and all else free.

(f) Find a simple upper bound on the average memory space required by the program, counting a recursive call as 1 location (for the return address) and all else free.

213 (Ackermann) Function *ack* of two natural variables is defined as follows.

$ack\ 0\ 0 = 2$
$ack\ 1\ 0 = 0$
$ack\ (m{+}2)\ 0 = 1$
$ack\ 0\ (n{+}1) = ack\ 0\ n + 1$
$ack\ (m{+}1)\ (n{+}1) = ack\ m\ (ack\ (m{+}1)\ n)$

(a) Suppose that functions and function application are not implemented expressions; in that case $n{:=}\ ack\ m\ n$ is not a program. Refine $n{:=}\ ack\ m\ n$ to obtain a program.

(b) Find a time bound. Hint: you may use function *ack* in your time bound.

(c) Find a space bound.

214 (alternate Ackermann) For each of the following functions f, refine $n{:=}\ f\ m\ n$, find a time bound (possibly involving f), and find a space bound.

(a)
$f\ 0\ n = n{+}2$
$f\ 1\ 0 = 0$
$f\ (m{+}2)\ 0 = 1$
$f\ (m{+}1)\ (n{+}1) = f\ m\ (f\ (m{+}1)\ n)$

(b)
$f\ 0\ n = n{\times}2$
$f\ (m{+}1)\ 0 = 1$
$f\ (m{+}1)\ (n{+}1) = f\ m\ (f\ (m{+}1)\ n)$

(c)
$f\ 0\ n = n{+}1$
$f\ 1\ 0 = 2$
$f\ 2\ 0 = 0$
$f\ (m{+}3)\ 0 = 1$
$f\ (m{+}1)\ (n{+}1) = f\ m\ (f\ (m{+}1)\ n)$

215 Let n be a natural variable. Add time according to the recursive measure, and find a finite upper bound on the execution time of

$P \Leftarrow$ **if** $n \geq 2$ **then** ($n{:=}\ n{-}2.\ P.\ n{:=}\ n{+}1.\ P.\ n{:=}\ n{+}1$) **else** *ok*

216√ (roller-coaster) Let n be a natural variable. It is easy to prove

$n'{=}1 \Leftarrow$ **if** $n{=}1$ **then** *ok*
else if *even n* **then** ($n{:=}\ n/2.\ n'{=}1$)
else ($n{:=}\ 3{\times}n + 1.\ n'{=}1$)

The problem is to find the execution time. Warning: this problem has never been solved.

217√ (Fibonacci) The Fibonacci numbers *fib n* are defined as follows.

$fib\ 0 = 0$
$fib\ 1 = 1$
$fib\ (n{+}2) = fib\ n + fib\ (n{+}1)$

Write a program to find *fib n* in time *log n*. Hint: see Exercise 301.

218 (Fibolucci) Let a and b be integers. Then the Fibolucci numbers for a and b are

$f\ 0 = 0$
$f\ 1 = 1$
$f\ (n{+}2) = a \times f\ n + b \times f\ (n{+}1)$

(The Fibonacci numbers are Fibolucci numbers for 1 and 1.) Given natural k, without using any list variables, write a program to compute

$\Sigma n{:}\ 0{,..}k{\cdot}\ fn \times f(k{-}n)$

219 (item count) Write a program to find the number of occurrences of a given item in a given list.

220 (duplicate count) Write a program to find how many items are duplicates (repeats) of earlier items
(a) in a given sorted nonempty list.
(b) in a given list.

221 (z-free subtext) Given a text, write a program to find the longest subtext that does not contain the letter "z".

222 (merge) Given two sorted lists, write a program to merge them into one sorted list.

223 (arithmetic) Let us represent a natural number as a list of naturals, each in the range $0,..b$ for some natural base $b>1$, in reverse order. For example, if $b=10$, then $[9; 2; 7]$ represents 729. Write programs for each of the following.
(a) Find the list representing a given natural in a given base.
(b) Given a base and two lists representing natural numbers, find the list representing their sum.
(c) Given a base and two lists representing natural numbers, find the list representing their difference. You may assume the first list represents a number greater than or equal to the number represented by the second list. What is the result if this is not so?
(d) Given a base and two lists representing natural numbers, find the list representing their product.
(e) Given a base and two lists representing natural numbers, find the lists representing their quotient and remainder.

224 (machine multiplication) Given two natural numbers, write a program to find their product using only addition, subtraction, doubling, halving, test for even, and test for zero.

225 (machine division) Given two natural numbers, write a program to find their quotient using only addition, subtraction, doubling, halving, test for even, and test for zero.

226 (machine squaring) Given a natural number, write a program to find its square using only addition, subtraction, doubling, halving, test for even, and test for zero.

227 Given a list of roots of a polynomial, write a program to find the list of coefficients.

228 (longest sorted sublist) Write a program to find the length of a longest sorted sublist of a given list, where
(a) the sublist must be consecutive items (a segment).
(b) the sublist consists of items in their order of appearance in the given list, but not necessarily consecutively.

229 (almost sorted segment) An almost sorted list is a list in which at most one adjacent pair of elements is out of order. Write a program to find the length of a longest almost sorted segment of a given list.

230 (edit distance) Given two lists, write a program to find the minimum number of item insertions, item deletions, and item replacements to change one list into the other.

231 (ultimately periodic sequence) You are given function f: $int \rightarrow int$ such that the sequence

$x_0 = 0$

$x_{n+1} = f(x_n)$

generated by f starting at 0 is ultimately periodic:

$\exists p$: $nat{+}1 \cdot \exists n$: $nat \cdot x_n = x_{n+p}$

The smallest positive p such that $\exists n$: $nat \cdot x_n = x_{n+p}$ is called the period. Write a program to find the period. Your program should use an amount of storage that is bounded by a constant, and not dependent on f.

232 (partitions) A list of positive integers is called a partition of natural number n if the sum of its items is n. Write a program to find

(a) a list of all partitions of a given natural n. For example, if n=3 then an acceptable answer is [[3]; [1; 2]; [2; 1]; [1; 1; 1]] .

(b) a list of all sorted partitions of a given natural n. For example, if n=3 then an acceptable answer is [[3]; [1; 2]; [1; 1; 1]] .

(c) the sorted list of all partitions of a given natural n. For example, if n=3 then the answer is [[1; 1; 1]; [1; 2]; [2; 1]; [3]] .

(d) the sorted list of all sorted partitions of a given natural n. For example, if n=3 then the answer is [[1; 1; 1]; [1; 2]; [3]] .

233 (largest true square) Write a program to find, within a boolean array, a largest square subarray consisting entirely of items with value T.

234 (P-list) Given a nonempty list S of natural numbers, define a P-list as a nonempty list P of natural numbers such that each item of P is an index of S, and

$\forall i$: $1,..\#P \cdot P\,(i{-}1) < P\,i \leq S\,(P\,(i{-}1))$

Write a program to find the length of a longest P-list for a given list S.

235 (J-list) For natural number n, a J-list of order n is a list of $2{\times}n$ naturals in which each m: $0,..n$ occurs twice, and between the two occurrences of m there are m items.

(a) Write a program that creates a J-list of order n if there is one, for given n.

(b) For which n do J-lists exist?

236 (diminished J-list) For positive integer n, a diminished J-list of order n is a list of $2{\times}n{-}1$ naturals in which 0 occurs once and each m: $1,..n$ occurs twice, and between the two occurrences of m there are m items.

(a) Write a program that creates a diminished J-list of order n if there is one, for given n.

(b) For which n do diminished J-lists exist?

237 (greatest common divisor) Given two positive integers, write a program to find their greatest common divisor.

238 (least common multiple) Given two positive integers, write a program to find their least common multiple.

239 Given two integers (not necessarily positive ones) that are not both zero, write a program to find their greatest common divisor.

240 (common items) Let A be a sorted list of different integers. Let B be another such list. Write a program to find the number of integers that occur in both lists.

241 (unique items) Let A be a sorted list of different integers. Let B be another such list. Write a program to find the sorted list of integers that occur in exactly one of A or B.

242 (smallest common item) Given two sorted lists having at least one item in common, write a program to find the smallest item occurring in both lists.

243 Given three sorted lists having at least one item common to all three, write a program to find the smallest item occurring in all three lists.

244 Given three positive integers, write a program to find their greatest common divisor. One method is to find the greatest common divisor of two of them, and then find the greatest common divisor of that and the remaining number, but there is a better way.

245 (longest common prefix) A natural number can be written as a sequence of decimal digits with a single leading zero. Given two natural numbers, write a program to find the number that is written as their longest common prefix of digits. For example, given 025621 and 02547, the result is 025. Hint: this question is about numbers, not about strings or lists.

246 (museum) You are given natural n, rationals s and f (start and finish), and lists A, D: [n*rat] (arrive and depart) such that

$$\forall i \cdot s \le Ai \le Di \le f$$

They represent a museum that opens at time s, is visited by n people with person i arriving at time Ai and departing at time Di and closes at time f. Write a program to find the total amount of time during which at least one person is inside the museum, and the average number of people in the museum during the time it is open, in time linear in n, if

(a) list A is sorted.

(b) list D is sorted.

247 (rotation test) Given two lists, write a program to determine if one list is a rotation of the other. You may use item comparisons, but not list comparisons. Execution time should be linear in the length of the lists.

248 (smallest rotation) Given a text variable t, write a program to reassign t its alphabetically (lexicographically) smallest rotation. You may use character comparisons, but not text comparisons.

249 You are given a list variable L assigned a nonempty list. All changes to L must be via procedure $swap$, defined as

$$swap = \langle i, j: 0,..\#L \rightarrow L := i \rightarrow Lj \mid j \rightarrow Li \mid L \rangle$$

(a) Write a program to reassign L a new list obtained by rotating the old list one place to the right (the last item of the old list is the first item of the new).

(b) (rotate) Given an integer r, write a program to reassign L a new list obtained by rotating the old list r places to the right. (If $r<0$, rotation is to the left $-r$ places.) Recursive execution time must be at most $\#L$.

(c) (segment swap) Given an index p, swap the initial segment up to p with the final segment beginning at p.

250 (squash) Let L be a list variable assigned a nonempty list. Reassign it so that any run of two or more identical items is collapsed to a single item.

251 Let n and p be natural variables. Write a program to solve

$$n \geq 2 \Rightarrow p' : 2^{2^{nat}} \wedge n \leq p' < n^2$$

Include a finite upper bound on the execution time, but it doesn't matter how small.

252 (greatest square under a histogram) You are given a histogram in the form of a list H of natural numbers. Write a program to find the longest segment of H in which the height (each item) is at least as large as the segment length.

253 (long texts) A particular computer has a hardware representation for texts of length n characters or less, for some constant n. Longer texts must be represented in software as a string of packaged short texts. (The long text represented is the catenation of the short texts.) A long text is called "packed" if all its items except possibly the last have length n. Write a program to pack a string of packaged short texts without changing the long text represented.

254 (Knuth, Morris, Pratt)

(a) Given list P, find list L such that for every index n of list P, Ln is the length of the longest list that is both a proper prefix and a proper suffix of $P\,[0;..n+1]$. Here is a program to find L.

$A \Leftarrow i := 0.\ L := [\#P * 0].\ j := 1.\ B$
$B \Leftarrow$ **if** $j \geq \#P$ **then** ok **else** $(C.\ L := j \rightarrow i \mid L.\ j := j+1.\ B)$
$C \Leftarrow$ **if** $Pi = Pj$ **then** $i := i+1$
else if $i = 0$ **then** ok
else $(i := L\,(i-1).\ C)$

Find specifications A, B, and C so that A is the problem and the three refinements are theorems.

(b) Given list S (subject), list P (pattern), and list L (as in part (a)), determine if P is a segment of S, and if so, where it occurs. Here is a program.

$D \Leftarrow m := 0.\ n := 0.\ E$
$E \Leftarrow$ **if** $m = \#P$ **then** $h := n - \#P$ **else** F
$F \Leftarrow$ **if** $n = \#S$ **then** $h := \infty$
else if $Pm = Sn$ **then** $(m := m+1.\ n := n+1.\ E)$
else G
$G \Leftarrow$ **if** $m = 0$ **then** $(n := n+1.\ F)$ **else** $(m := L\,(m-1).\ G)$

Find specifications D, E, F, and G so that D is the problem and the four refinements are theorems.

End of Program Theory

10.5 Programming Language

255 (nondeterministic assignment) Generalize the assignment notation $x := e$ to allow the expression e to be a bunch, with the meaning that x is assigned an arbitrary element of the bunch. For example, $x := nat$ assigns x an arbitrary natural number. Show the standard boolean notation for this form of assignment. Show what happens to the Substitution Law.

256 Suppose variable declaration is defined as

var x: $T \cdot P \quad = \quad \exists x$: $undefined \cdot \exists x'$: $T \cdot P$

What are the characteristics of this kind of declaration? Look at the example

var x: $int \cdot ok$

257 What is wrong with defining local variable declaration as follows:

var x: T· P = $\forall x$: T· $\exists x'$: T· P

258 Suppose variable declaration with initialization is defined as

var x: T := e· P = **var** x: T· x:= e. P

In what way does this differ from the definition given in Subsection 5.0.0?

259 Here are two different definitions of variable declaration with initialization.

var x: T := e· P = $\exists x, x'$: T· x=e ∧ P

var x: T := e· P = $\exists x'$: T· (substitute e for x in P)

Show how they differ with an example.

260 The specification

var x: *nat*· x:= –1

introduces a local variable and then assigns it a value that is out of bounds. Is this specification implementable? (Proof required.)

261 (frame problem) Suppose there is one nonlocal variable x, and we define P = x'=0. Can we prove

P ⇐ **var** y: *nat*· y:= 0. P. x:= y

The problem is that y was not part of the state space where P was defined, so does P leave y unchanged? Hint: consider the definition of dependent composition. Is it being used properly?

262 Let the state variables be x, y, and z. Rewrite **frame** x· ⊤ without using **frame**. Say in words what the final value of x is.

263 In a language with array element assignment, the program

x:= i. i:= A i. A i:= x

was written with the intention to swap the values of i and A i. Assume that all variables and array elements are of type *nat*, and that i has a value that is an index of A.

(a) In variables x, i, and A, specify that i and A i should be swapped, the rest of A should be unchanged, but x might change.

(b) Find the exact precondition for which the program refines the specification of part (a).

(c) Find the exact postcondition for which the program refines the specification of part (a).

264 In a language with array element assignment, what is the exact precondition for A' i' = 1 to be refined by (A(A i):= 0. A i:= 1. i:= 2) ?

265√ (unbounded bound) Find a time bound for the following program in natural variables x and y.

while ¬ x=y=0 **do**
  if y>0 **then** y:= y–1
  else (x:= x–1. **var** n: *nat*· y:= n)

266 Let W ⇐ **while** b **do** P be an abbreviation of W ⇐ **if** b **then** (P. W) **else** *ok*. Let R ⇐ **repeat** P **until** b be an abbreviation of R ⇐ P. **if** b **then** *ok* **else** R. Now prove

(R ⇐ **repeat** P **until** b) ∧ (W ⇐ **while** ¬b **do** P)
⇐ (R ⇐ P. W) ∧ (W ⇐ **if** b **then** *ok* **else** R)

267 (guarded command) In "Dijkstra's little language" there is a conditional program with the syntax

if $b \to P$ [] $c \to Q$ **fi**

where b and c are boolean and P and Q are programs. It can be executed as follows. If exactly one of b and c is true initially, then the corresponding program is executed; if both b and c are true initially, then either one of P or Q (arbitrary choice) is executed; if neither b nor c is true initially, then execution is completely arbitrary.

(a) Express this program in the notations of this book as succinctly as possible.

(b) Refine this program using only the programming notations introduced in Chapter 4.

268√ Using a **for**-loop, write a program to add 1 to every item of a list.

269 Here is one way that we might consider defining the **for**-loop. Let j, n, k and m be integer expressions, and let i be a fresh name.

for i:= *nil* **do** P = *ok*

for i:= j **do** P = (substitute j for i in P)

for i:= n;..k ; k;..m **do** P = **for** i:= n;..k **do** P. **for** i:= k;..m **do** P

(a) From this definition, what can we prove about **for** i:= 0;..n **do** n:= n+1 where n is an integer variable?

(b) What kinds of **for**-loop are in the programming languages you know?

270 (majority vote) The problem is to find, in a given list, the majority item (the item that occurs in more than half the places) if there is one. Letting L be the list and m be a variable whose final value is the majority item, prove that the following program solves the problem.

(a)
```
var e: nat := 0·
for i:= 0;..#L do
    if m = L i then e:= e+1
    else if i = 2×e then (m:= L i. e:= e+1)
    else ok
```

(b)
```
var s: nat := 0·
for i:= 0;..#L do
    if m = L i then ok
    else if i = 2×s then m:= L i
    else s:= s+1
```

271 We defined the programmed expression P **result** e with the axiom

$x' = (P \textbf{ result } e) = P.\ x'=e$

Why don't we define it instead with the axiom

$x' = (P \textbf{ result } e) = P \Rightarrow x'=e'$

272 Let a and b be rational variables. Define procedure P as

$P = \langle x, y: rat \to \textbf{if } x=0 \textbf{ then } a:= x \textbf{ else } (a:= x\times y.\ a:= a\times y)\rangle$

(a) What is the exact precondition for $a'=b'$ to be refined by $P\ a\ (1/b)$?

(b) Discuss the difference between "eager" and "lazy" evaluation of arguments as they affect both the theory of programming and programming language implementation.

273 "Call-by-value-result" describes a parameter that gets its initial value from an argument, is then a local variable, and gives its final value back to the argument, which therefore must be a variable. Define "call-by-value-result" formally. Discuss its merits and demerits.

274 (call-by-name) Here is a procedure applied to an argument.

$\langle x$: *int*→a:= x. b:= $x\rangle$ (a+1)

Suppose, by mistake, we replace both occurrences of x in the body with the argument. What do we get? What should we get? (This mistake is known as "call-by-name".)

275 We defined **wait until** w = t:= *max* t w where t is an extended integer time variable, and w is an integer expression.

(a)√ Prove **wait until** w ⇐ **if** $t \geq w$ **then** *ok* **else** (t:= t+1. **wait until** w)

(b) Now suppose that t is an extended real time variable, and w is an extended real expression. Redefine **wait until** w appropriately, and refine it using the real time measure (assume any positive operation time you need).

276 The specification **wait** w where w is a length of time, not an instant of time, describes a delay in execution of time w. Formalize and implement it using

(a) the recursive time measure.

(b) the real time measure (assume any positive operation times you need).

277 We propose to define a new programming connective $P \blacklozenge Q$. What properties of ♦ are essential? Why?

278 (Boole's booleans) If ⊤=1 and ⊥=0, express

(a) $\neg a$

(b) $a \wedge b$

(c) $a \vee b$

(d) $a \Rightarrow b$

(e) $a \Leftarrow b$

(f) $a = b$

(g) $a \neq b$

using only the following symbols (in any quantity)

(i) 0 1 a b ( ) + − ×

(ii) 0 1 a b ( ) − *max min*

279 Prove that the average value of

(a) n^2 as n varies over *nat*+1 according to probability 2^{-n} is 6.

(b) n as it varies over *nat* according to probability $(5/6)^n \times 1/6$ is 5.

280 (coin) Repeatedly flip a coin until you get a head. Prove that it takes n flips with probability 2^{-n}. With an appropriate definition of R, the program is

R ⇐ t:= t+1. **if** *rand* 2 **then** *ok* **else** R

281 (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,

(a)√ find the probability for each value of your total.

(b) find the average value of your total.

282√ (dice) If you repeatedly throw a pair of six-sided dice until they are equal, how long does it take?

283 (drunk) A drunkard is trying to walk home. At each time unit, the drunkard may go forward one distance unit, stay in the same position, or go back one distance unit. After n time units, where is the drunkard?

(a) At each time unit, there is 2/3 probability of going forward, and 1/3 probability of staying in the same position. The drunkard does not go back.

(b) At each time unit, there is 1/4 probability of going forward, 1/2 probability of staying in the same position, and 1/4 probability of going back.

(c) At each time unit, there is 1/2 probability of going forward, 1/4 probability of staying in the same position, and 1/4 probability of going back.

284 (Mr.Bean's socks) Mr.Bean is trying to get a matching pair of socks from a drawer containing an inexhaustible supply of red and blue socks. He begins by withdrawing two socks at random. If they match, he is done. Otherwise, he throws away one of them at random, withdraws another sock at random, and repeats. How long will it take him to get a matching pair? Assume that a sock withdrawn from the drawer has 1/2 probability of being each color, and that a sock that is thrown away also has a 1/2 probability of being each color.

End of Programming Language

10.6 Recursive Definition

285 Prove $\neg -1: nat$. Hint: You will need induction.

286 (Cantor's diagonal) Prove $\neg \exists f: nat \to nat \to nat \cdot \forall g: nat \to nat \cdot \exists n: nat \cdot fn = g$.

287 Prove $\forall n: nat \cdot Pn \;=\; \forall n: nat \cdot \forall m: 0,..n \cdot Pm$

288√ Prove that the square of an odd natural number is $8 \times m + 1$ for some natural m .

289 Prove that every positive integer is a product of primes. By "product" we mean the result of multiplying together any natural number of (not necessarily distinct) numbers. By "prime" we mean a natural number with exactly two factors.

290 Here is an argument to "prove" that in any group of people, all the people are the same age. The "proof" is by induction on the size of groups. The induction base is that in any group of size 1 , clearly all the people are the same age. Or we could equally well use groups of size 0 as the induction base. The induction hypothesis is, of course, to assume that in any group of size n , all the people are the same age. Now consider a group of size $n+1$. Let its people be $p_0, p_1, ..., p_n$. By the induction hypothesis, in the subgroup $p_0, p_1, ..., p_{n-1}$ of size n , all the people are the same age; to be specific, they are all the same age as p_1 . And in the subgroup $p_1, p_2, ..., p_n$ of size n , all the people are the same age; again, they are the same age as p_1 . Hence all $n+1$ people are the same age. Formalize this argument and find the flaw.

291 Here is a possible alternative construction axiom for nat .

$0, 1, nat+nat: nat$

(a) What induction axiom goes with it?

(b) Are the construction axiom given and your induction axiom of part (a) satisfactory as a definition if nat?

292 Chapter 6 gives four predicate versions of *nat* induction. Prove that they are equivalent.

293 Prove $nat = 0,..\infty$.

294 Here are a construction axiom and an induction axiom for bunch *bad* .

$(\S n\colon nat\cdot \neg\ n\colon bad) : bad$

$(\S n\colon nat\cdot \neg\ n\colon B) : B \implies bad\colon B$

(a)√ Are these axioms consistent?

(b) From these axioms, can we prove the fixed-point equation

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

295 Prove the following; quantifications are over *nat* .

(a) $\neg\exists i,j\cdot\ j{\neq}0\ \wedge\ 2^{1/2} = i/j$ The square root of 2 is irrational.

(b) $\forall n\cdot\ (\Sigma i\colon 0,..n\cdot\ 1) = n$

(c) $\forall n\cdot\ (\Sigma i\colon 0,..n\cdot\ i) = n \times (n{-}1) / 2$

(d) $\forall n\cdot\ (\Sigma i\colon 0,..n\cdot\ i^3) = (\Sigma i\colon 0,..n\cdot\ i)^2$

(e) $\forall n\cdot\ (\Sigma i\colon 0,..n\cdot\ 2^i) = 2^n - 1$

(f) $\forall n\cdot\ (\Sigma i\colon 0,..n\cdot\ i{\times}2^i) = (n{-}2){\times}2^n + 2$

(g) $\forall n\cdot\ (\Sigma i\colon 0,..n\cdot\ (-2)^i) = (1 - (-2)^n) / 3$

(h) $\forall n\cdot\ n{\geq}10 \implies 2^n > n^3$

(i) $\forall n\cdot\ n{\geq}4 \implies 3^n > n^3$

(j) $\forall n\cdot\ n{\geq}3 \implies 2{\times}n^3 > 3{\times}n^2 + 3{\times}n$

(k) $\forall a,d\cdot\ \exists q,r\cdot\ d{\neq}0 \implies r{<}d\ \wedge\ a = q{\times}d + r$

(l) $\forall a,b\cdot\ a{\leq}b \implies (\Sigma i\colon a,..b\cdot\ 3^i) = (3^b{-}3^a)/2$

296 Show that we can define *nat* by fixed-point construction together with

(a) $\forall n\colon nat\cdot\ 0 \leq n < n{+}1$

(b) $\exists m\colon nat\cdot\ \forall n\colon nat\cdot\ m \leq n < n{+}1$

297√ Suppose we define *nat* by ordinary construction and induction.

$0, nat{+}1\colon nat$

$0, B{+}1\colon B \implies nat\colon B$

Prove that fixed-point construction and induction

$nat = 0, nat{+}1$

$B = 0, B{+}1 \implies nat\colon B$

are theorems.

298 (fixed-point theorem) Suppose we define *nat* by fixed-point construction and induction.

$nat = 0, nat{+}1$

$B = 0, B{+}1 \implies nat\colon B$

Prove that ordinary construction and induction

$0, nat{+}1\colon nat$

$0, B{+}1\colon B \implies nat\colon B$

are theorems. Warning: this is hard, and requires the use of limits.

299 (rulers) Rulers are formed as follows. A vertical stroke | is a ruler. If you append a horizontal stroke — and then a vertical stroke | to a ruler you get another ruler. Thus the first few rulers are | , |—| , |—|—| , |—|—|—| , and so on. No two rulers formed this way are equal. There are no other rulers. What axioms are needed to define bunch *ruler* consisting of all and only the rulers?

300 Function f is called monotonic if $\forall i,j\cdot\ i \le j \Rightarrow f\,i \le f\,j$.

(a) Prove f is monotonic if and only if $\forall i,j\cdot\ f\,i < f\,j \Rightarrow i < j$.

(b) Let f: $int \to int$. Prove f is monotonic if and only if $\forall i\cdot\ f\,i \le f\,(i{+}1)$.

(c) Let f: $nat \to nat$ be such that $\forall n\cdot f\,f\,n < f\,(n{+}1)$. Prove f is the identity function. Hints: First prove $\forall n\cdot\ n \le f\,n$. Then prove f is monotonic. Then prove $\forall n\cdot f\,n \le n$.

301 The Fibonacci numbers $fib\ n$ are defined as follows.

$$fib\ 0 = 0$$
$$fib\ 1 = 1$$
$$fib\ (n{+}2) = fib\ n + fib\ (n{+}1)$$

Prove

(a) $$fib\ (gcd\ n\ m) = gcd\ (fib\ n)\ (fib\ m)$$

where gcd is the greatest common divisor.

(b) $$fib\ n \times fib\ (n{+}2) = fib\ (n{+}1)^{\,2} - (-1)^n$$

(c) $$fib\ (n{+}m{+}1) = fib\ n \times fib\ m + fib\ (n{+}1) \times fib\ (m{+}1)$$

(d) $$fib\ (n{+}m{+}2) = fib\ n \times fib\ (m{+}1) + fib\ (n{+}1) \times fib\ m + fib\ (n{+}1) \times fib\ (m{+}1)$$

(e) $$fib\ (2{\times}n{+}1) = fib\ n^{\,2} + fib\ (n{+}1)^{\,2}$$

(f) $$fib\ (2{\times}n{+}2) = 2 \times fib\ n \times fib\ (n{+}1) + fib\ (n{+}1)^{\,2}$$

302 Let R be a relation of naturals R: $nat \to nat \to bool$ that is monotonic in its second parameter

$$\forall i,j\cdot\ R\ i\ j \Rightarrow R\ i\ (j{+}1)$$

Prove

$$\exists i\cdot\ \forall j\cdot\ R\ i\ j\ =\ \forall j\cdot\ \exists i\cdot\ R\ i\ j$$

303 What is the smallest bunch satisfying

(a) $$B = 0, 2{\times}B + 1$$

(b) $$B = 2, B{\times}B$$

304 What elements can be proven in P from the axiom $P = 1, x, -P, P{+}P, P{\times}P$? Prove

$$2{\times}x^2{-}1\colon P$$

305 Bunch $this$ is defined by the construction and induction axioms

$$2, 2{\times}this\colon\ this$$
$$2, 2{\times}B\colon\ B \Rightarrow this\colon B$$

Bunch $that$ is defined by the construction and induction axioms

$$2, that{\times}that\colon\ that$$
$$2, B{\times}B\colon\ B \Rightarrow that\colon B$$

Prove $this = that$.

306 Express 2^{int} without using exponentiation. You may introduce auxiliary names.

307 Let n be a natural number. From the fixed-point equation

$$ply = n, ply{+}ply$$

we obtain a sequence of bunches ply_i by recursive construction.

(a) State ply_i formally (no proof needed).

(b) State ply_i in English.

(c) What is ply_∞ ?

(d) Is ply_∞ a solution? If so, is it the only solution?

308 For each of the following fixed-point equations, what does recursive construction yield? Does it satisfy the fixed-point equation?

(a) $M = [*int], [*M]$

(b) $T = [nil], [T; int; T]$

(c) $A = bool, rat, char, [*A]$

309 Let $A \setminus B$ be the difference between bunch A and bunch B. The operator $\setminus$ has precedence level 4, and is defined by the axiom

$x: A \setminus B \;=\; x: A \;\wedge\; \neg\, x: B$

For each of the following fixed-point equations, what does recursive construction yield? Does it satisfy the fixed-point equation?

(a) $Q = nat \setminus (Q+3)$

(b) $D = 0, (D+1) \setminus (D-1)$

(c) $E = nat \setminus (E+1)$

(d) $F = 0, (nat \setminus F)+1$

310 For each of the following fixed-point equations, what does recursive construction yield? Does it satisfy the fixed-point equation?

(a) $P = \S n: nat \cdot n=0 \wedge P=null \;\vee\; n: P+1$

(b) $Q = \S x: xnat \cdot x=0 \wedge Q=null \;\vee\; x: Q+1$

311 Here is a pair of mutually recursive equations.

$even = 0, odd+1$

$odd = even+1$

(a) What does recursive construction yield? Show the construction.

(b) Are further axioms needed to ensure that *even* consists of only the even naturals, and *odd* consists of only the odd naturals? If so, what axioms?

312(a) Considering E as the unknown, find three solutions of $E, E+1 = nat$.

(b) Now add the induction axiom $B, B+1 = nat \;\Rightarrow\; E: B$. What is E?

313 From the construction axiom $0, 1–few: few$

(a) what elements are constructed?

(b) give three solutions (considering *few* as the unknown).

(c) give the corresponding induction axiom.

(d) state which solution is specified by construction and induction.

314 Investigate the fixed-point equation

$strange = \S n: nat \cdot \forall m: strange \cdot \neg\, m+1: n \times nat$

315 Let *truer* be a bunch of strings of booleans defined by the construction and induction axioms

$\mathsf{T}, \bot; truer; truer: truer$

$\mathsf{T}, \bot; B; B: B \Rightarrow truer: B$

Given a string of booleans, write a program to determine if the string is in *truer*.

316 (strings) If S is a bunch of strings, then $*S$ is the bunch of all strings formed by catenating together any number of any strings in S in any order. Define $*S$ by construction and induction.

317 Here are the construction and induction axioms for lists of items of type T.

$[nil], [T], list+list: list$

$[nil], [T], L+L: L \implies list: L$

Prove $list = [*T]$.

318 (decimal-point numbers) Using recursive data definition, define the bunch of all decimal-point numbers. These are the rationals that can be expressed as a finite string of decimal digits containing a decimal point. Note: you are defining a bunch of numbers, not a bunch of texts.

319 (Backus-Naur Form) Backus-Naur Form is a grammatical formalism in which grammatical rules are written as in the following example.

$\langle exp \rangle ::= \langle exp \rangle + \langle exp \rangle \mid \langle exp \rangle \times \langle exp \rangle \mid 0 \mid 1$

In our formalism, it would be written

$exp \;=\; exp; \text{"+"}; exp, \;\; exp; \text{"×"}; exp, \;\; \text{"0"}, \;\; \text{"1"}$

In a similar fashion, write axioms to define each of the following.

(a) palindromes: texts that read the same forward and backward. Use a two-symbol alphabet.

(b) palindromes of odd length.

(c) all texts consisting of "a"s followed by the same number of "b"s.

(d) all texts consisting of "a"s followed by at least as many "b"s.

320 Section 6.1 defines program *zap* by the fixed-point equation

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

(a) Prove $zap \implies x{\geq}0 \implies x'{=}y'{=}0 \wedge t' = t{+}x$.

(b) Prove $x{\geq}0 \wedge x'{=}y'{=}0 \wedge t' = t{+}x \implies zap$.

(c) What axiom is needed to make *zap* the weakest fixed-point?

(d) What axiom is needed to make *zap* the strongest fixed-point?

(e) Section 6.1 gives six solutions to this equation. Find more solutions. Hint: strange things can happen at time ∞.

321 Let all variables be integer. Add recursive time. Using recursive construction, find a fixed-point of

(a) $skip \;=\; \textbf{if } i{\geq}0 \textbf{ then } (i{:=}\, i{-}1.\;\; skip.\;\; i{:=}\, i{+}1) \textbf{ else } ok$

(b) $inc \;=\; ok \vee (i{:=}\, i{+}1.\;\; inc)$

(c) $sqr \;=\; \textbf{if } i{=}0 \textbf{ then } ok \textbf{ else } (s{:=}\, s + 2{\times}i - 1.\;\; i{:=}\, i{-}1.\;\; sqr)$

(d) $fac \;=\; \textbf{if } i{=}0 \textbf{ then } f{:=}\, 1 \textbf{ else } (i{:=}\, i{-}1.\;\; fac.\;\; i{:=}\, i{+}1.\;\; f{:=}\, f{\times}i)$

(e) $chs \;=\; \textbf{if } a{=}b \textbf{ then } c{:=}\, 1 \textbf{ else } (a{:=}\, a{-}1.\;\; chs.\;\; a{:=}\, a{+}1.\;\; c{:=}\, c{\times}a/(a{-}b))$

322 Let all variables be integer. Add recursive time. Any way you can, find a fixed-point of

(a) $walk \;=\; \textbf{if } i{\geq}0 \textbf{ then } (i{:=}\, i{-}2.\;\; walk.\;\; i{:=}\, i{+}1.\;\; walk.\;\; i{:=}\, i{+}1) \textbf{ else } ok$

(b) $crawl \;=\; \textbf{if } i{\geq}0 \textbf{ then } (i{:=}\, i{-}1.\;\; crawl.\;\; i{:=}\, i{+}2.\;\; crawl.\;\; i{:=}\, i{-}1) \textbf{ else } ok$

(c) $run \;=\; \textbf{if } even\ i \textbf{ then } i{:=}\, i/2 \textbf{ else } i{:=}\, i{+}1.$
$\textbf{if } i{=}1 \textbf{ then } ok \textbf{ else } run$

323 Investigate how recursive construction is affected when we start with

(a) $t' = \infty$

(b) $t{:=}\, \infty$

324 Let x be an integer variable. Using the recursive time measure, add time and then find the strongest implementable specifications P and Q that you can find for which

$$P \Leftarrow x' \geq 0.\ Q$$
$$Q \Leftarrow \textbf{if } x{=}0 \textbf{ then } ok \textbf{ else } (x{:=}\ x{-}1.\ Q)$$

Assume that $x' \geq 0$ takes no time.

325 Let x be an integer variable.

(a) Using the recursive time measure, add time and then find the strongest implementable specification S that you can find for which

$$S \Leftarrow \textbf{if } x{=}0 \textbf{ then } ok \textbf{ else if } x{>}0 \textbf{ then } (x{:=}\ x{-}1.\ S) \textbf{ else } (x' \geq 0.\ S)$$

Assume that $x' \geq 0$ takes no time.

(b) What do we get from recursive construction starting with $t' \geq t$?

326 Prove that the following three ways of defining R are equivalent.

$$R = ok \vee (R.\ S)$$
$$R = ok \vee (S.\ R)$$
$$R = ok \vee S \vee (R.\ R)$$

327 Prove the laws of Refinement by Steps and Refinement by Parts for **while**-loops.

328 Prove that

$$\forall \sigma, \sigma' \cdot (t' \geq t \wedge (\textbf{if } b \textbf{ then } (P.\ t{:=}\ t{+}1.\ W) \textbf{ else } ok) \Leftarrow W) \Leftarrow \forall \sigma, \sigma' \cdot (\textbf{while } b \textbf{ do } P \Leftarrow W)$$

is equivalent to the **while** construction axioms, and hence that construction and induction can be expressed together as

$$\forall \sigma, \sigma' \cdot (t' \geq t \wedge (\textbf{if } b \textbf{ then } (P.\ t{:=}\ t{+}1.\ W) \textbf{ else } ok) \Leftarrow W) = \forall \sigma, \sigma' \cdot (\textbf{while } b \textbf{ do } P \Leftarrow W)$$

329 The notation **do** P **while** b has been used as a loop construct that is executed as follows. First P is executed; then b is evaluated, and if $\top$ execution is repeated, and if $\bot$ execution is finished. Define **do** P **while** b by construction and induction axioms.

330 Using the definition of Exercise 329, but ignoring time, prove

(a) $\textbf{do } P \textbf{ while } b = P.\ \textbf{while } b \textbf{ do } P$

(b) $\textbf{while } b \textbf{ do } P = \textbf{if } b \textbf{ then do } P \textbf{ while } b \textbf{ else } ok$

(c) $(\forall \sigma, \sigma' \cdot (D = \textbf{do } P \textbf{ while } b)) \wedge (\forall \sigma, \sigma' \cdot (W = \textbf{while } b \textbf{ do } P)) = (\forall \sigma, \sigma' \cdot (D = P.\ W)) \wedge (\forall \sigma, \sigma' \cdot (W = \textbf{if } b \textbf{ then } D \textbf{ else } ok))$

331 Let P: $nat \rightarrow bool$.

(a) Define quantifier $FIRST$ so that $FIRST\ m{:}\ nat \cdot Pm$ is the smallest natural m such that Pm , and ∞ if there is none.

(b) Prove $n{:=}\ FIRST\ m{:}\ nat \cdot Pm \Leftarrow n{:=}\ 0.\ \textbf{while } \neg Pn \textbf{ do } n{:=}\ n{+}1$.

332 Let the state consist of boolean variables b and c . Let

$$W = \textbf{if } b \textbf{ then } (P.\ W) \textbf{ else } ok$$
$$X = \textbf{if } b \vee c \textbf{ then } (P.\ X) \textbf{ else } ok$$

(a) Find a counterexample to $W.\ X = X$.

(b) Now let W and X be the weakest solutions of those equations, and prove $W.\ X = X$.

333 In real variable x, consider the equation

$$P = P.\ x := x^2$$

(a) Find 7 distinct solutions for P.

(b) Which solution does recursive construction give starting from $\top$? Is it the weakest solution?

(c) If we add a time variable, which solution does recursive construction give starting from $t' \geq t$? Is it a strongest implementable solution?

(d) Now let x be an integer variable, and redo the question.

334 Suppose we define **while** b **do** P by ordinary construction and induction, ignoring time.

$$\textbf{if } b \textbf{ then } (P.\ \textbf{while } b \textbf{ do } P) \textbf{ else } ok \Leftarrow \textbf{while } b \textbf{ do } P$$

$$\forall \sigma, \sigma' \cdot (\textbf{if } b \textbf{ then } (P.\ W) \textbf{ else } ok \Leftarrow W) \Rightarrow \forall \sigma, \sigma' \cdot (\textbf{while } b \textbf{ do } P \Leftarrow W)$$

Prove that fixed-point construction and induction

$$\textbf{while } b \textbf{ do } P = \textbf{if } b \textbf{ then } (P.\ \textbf{while } b \textbf{ do } P) \textbf{ else } ok$$

$$\forall \sigma, \sigma' \cdot (W = \textbf{if } b \textbf{ then } (P.\ W) \textbf{ else } ok) \Rightarrow \forall \sigma, \sigma' \cdot (\textbf{while } b \textbf{ do } P \Leftarrow W)$$

are theorems.

335 Suppose we define **while** b **do** P by fixed-point construction and induction, ignoring time.

$$\textbf{while } b \textbf{ do } P = \textbf{if } b \textbf{ then } (P.\ \textbf{while } b \textbf{ do } P) \textbf{ else } ok$$

$$\forall \sigma, \sigma' \cdot (W = \textbf{if } b \textbf{ then } (P.\ W) \textbf{ else } ok) \Rightarrow \forall \sigma, \sigma' \cdot (\textbf{while } b \textbf{ do } P \Leftarrow W)$$

Prove that ordinary construction and induction

$$\textbf{if } b \textbf{ then } (P.\ \textbf{while } b \textbf{ do } P) \textbf{ else } ok \Leftarrow \textbf{while } b \textbf{ do } P$$

$$\forall \sigma, \sigma' \cdot (\textbf{if } b \textbf{ then } (P.\ W) \textbf{ else } ok \Leftarrow W) \Rightarrow \forall \sigma, \sigma' \cdot (\textbf{while } b \textbf{ do } P \Leftarrow W)$$

are theorems. Warning: this is hard, and requires the use of limits.

End of Recursive Definition

10.7 Theory Design and Implementation

336 (widgets) A theory of widgets is presented in the form of some new syntax and some axioms. An implementation of widgets is written.

(a) How do we know whether the theory of widgets is consistent or inconsistent?

(b) How do we know whether the theory of widgets is complete or incomplete?

(c) How do we know whether the implementation of widgets is correct or incorrect?

337√ Implement data-stack theory to make the two boolean expressions

$$pop\ empty = empty$$

$$top\ empty = 0$$

antitheorems.

338 Prove that the following definitions implement the simple data-stack theory.

$$stack = [nil], [stack; X]$$

$$push = \langle s: stack \rightarrow \langle x: X \rightarrow [s; x] \rangle \rangle$$

$$pop = \langle s: stack \rightarrow s\ 0 \rangle$$

$$top = \langle s: stack \rightarrow s\ 1 \rangle$$

339 (weak data-stack) In Subsection 7.1.3 we designed a program-stack theory so weak that we could add axioms to count pushes and pops without inconsistency. Design a similarly weak data-stack theory.

340 (data-queue implementation) Implement the data-queue theory presented in Section 7.0.

341 (slip) The slip data structure introduces the name *slip* with the following axioms:

$$slip = [X; slip]$$

$$B = [X; B] \Rightarrow B: slip$$

where X is some given type. Can you implement it?

342 Prove that the program-stack implementation given in Subsection 7.1.1 satisfies the program-stack axioms of Subsection 7.1.0.

343 Implement weak program-stack theory as follows: the implementer's variable is a list that grows and never shrinks. A popped item must be marked as garbage.

344 You are given a program-stack. Can you write a program composed from the programs

push "A" *push* "B" *push* "C" *push* "D" *push* "E"

in that order, with the programs *print top* and *pop* interspersed wherever needed as many times as needed, to obtain the following output?

(a) B D E C A
(b) B C D E A
(c) C A D E B
(d) A B E C D
(e) A B C D E

345 (brackets) You are given a text t of characters drawn from the alphabet "x", "(", ")", "[", "]" . Write a program to determine if t has its brackets properly paired and nested.

346 (limited-stack) A stack, according to our axioms, has an unlimited capacity to have items pushed onto it. A limited-stack is a similar data structure but with a limited capacity to have items pushed onto it.

(a) Design axioms for a limited-data-stack.
(b) Design axioms for a limited-program-stack.
(c) Can the limit be 0 ?

347 (limited-queue) A queue, according to our axioms, has an unlimited capacity to have items joined onto it. A limited-queue is a similar data structure but with a limited capacity to have items joined onto it.

(a) Design axioms for a limited-data-queue.
(b) Design axioms for a limited-program-queue.
(c) Can the limit be 0 ?

348 You are given a program-queue. Can you write a program composed from the programs

join "A" *join* "B" *join* "C" *join* "D" *join* "E"

in that order, with the programs *print front* and *leave* interspersed wherever needed as many times as needed, to obtain the following output?

(a) B D E C A
(b) B C D E A
(c) C A D E B
(d) A B E C D
(e) A B C D E

349 Each of the program theories provides a single, anonymous instance of a data structure. How can a program theory be made to provide many instances of a data structure, like data theories do?

350 (circular list) Design axioms for a circular list. There should be operations to create an empty list, to move along one position in the list (the first item comes after the last, in circular fashion), to insert an item at the current position, to delete the current item, and to give the current item.

351 (resettable variable) A resettable variable is defined as follows. There are three new names: *value* (of type X), *set* (a procedure with one parameter of type X), and *reset* (a program). Here are the axioms:

$value'=x \Leftarrow set\ x$

$value'=value \Leftarrow set\ x.\ reset$

$reset.reset = reset$

Implement this data structure, with proof.

352 A particular program-list has the following operations:

- the operation *mkempty* makes the list empty
- the operation *extend x* catenates item x to the end of the list
- the operation *swap i j* swaps the items at indexes i and j
- the expression *length* tells the length of the list
- the expression *item i* tells the item at index i

(a) Write axioms to define this program-list.
(b) Implement this program-list, with proof.

353 (linear algebra) Design a theory of linear algebra. It should include scalar, vector, and matrix sums, products, and inner products. Implement the theory, with proof.

354 (leafy tree) A leafy tree is a tree with information residing only at the leaves. Design appropriate axioms for a binary leafy data-tree.

355 A tree can be implemented by listing its items in breadth order.
(a) Implement a binary tree by a list of its items such that the root is at index 0 and the left and right subtrees of an item at index n are rooted at indexes $2\times n+1$ and $2\times n+2$.
(b) Prove your implementation.
(c) Generalize this implementation to trees in which each item can have at most k branches for arbitrary (but constant) k.

356 (hybrid-tree) Chapter 7 presented data-tree theory and program-tree theory. Design a hybrid-tree theory in which there is only one tree structure, so it can be an implementer's variable with program operations on it, but there can be many pointers into the tree, so they are data-pointers (they may be data-stacks).

357 (heap) A heap is a tree with the property that the root is the largest item and the subtrees are heaps.
(a) Specify the heap property formally.
(b) Write a function *heapgraft* that makes a heap from two given heaps and a new item. It may make use of *graft*, and may rearrange the items as necessary to produce a heap.

358 (leaf count) Write a program to count the number of leaves in a tree.

359 (binary search tree) A binary search tree is a binary tree with the property that all items in the left subtree are less than the root item, all items in the right subtree are greater than the root item, and the subtrees are also binary search trees.

(a) Specify the binary search tree property formally.
(b) How many binary search trees are there with three items?
(c) How many binary search trees are there with three distinct items?
(d) Write a program to find an item in a binary search tree.
(e) Write a program to add an item to a binary search tree as a new leaf.
(f) Write a program to make a list of the items in a binary search tree in order.
(g) Write a program to determine whether two binary search trees have the same items.

360 (party) A company is structured as a tree, with employees at the nodes. Each employee, except the one at the root, has a boss represented by their parent in the tree. Each employee has a conviviality rating (a number) representing how much fun they are at a party. But no-one will be at a party with their boss. Write a program to find the bunch of employees to invite to a party so that the total convivialty is maximized.

361 (insertion list) An insertion list is a data structure similar to a list, but with an associated insertion point.

[...; 4 ; 7 ; 1 ; 0 ; 3 ; 8 ; 9 ; 2 ; 5 ; ...]
↑
insertion point

insert puts an item at the insertion point (between two existing items), leaving the insertion point at its right. *erase* removes the item to the left of the insertion point, closing up the list. *item* gives the item to the left of the insertion point. *forward* moves the insertion point one item to the right. *back* moves the insertion point one item to the left.

(a) Design axioms for a doubly-infinite data-insertion list.
(b) Design axioms for a doubly-infinite program-insertion list.
(c) Design axioms for a finite data-insertion list.
(d) Design axioms for a finite program-insertion list.

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.

363 A theory provides three names: *zero* , *increase* , and *inquire* . It is presented by an implementation. Let u: *bool* be the user's variable, and let v: *nat* be the implementer's variable. The axioms are

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

Use data transformation to replace v with w: *bool* according to the transformer

(a)√ w = *even* v
(b) ⊤
(c) ⊥ (this isn't a data transformer, since $\forall w \cdot \exists v \cdot \bot$ isn't a theorem, but apply it anyway to see what happens)

364 A theory provides three names: *set* , *flip* , and *ask* . It is presented by an implementation. Let u: *bool* be the user's variable, and let v: *bool* be the implementer's variable. The axioms are

$$set = v := \top$$
$$flip = v := \neg v$$
$$ask = u := v$$

(a)√ Replace v with w: *nat* according to the data transformer $v = even\ w$.

(b) Replace v with w: *nat* according to the data transformer $(w{=}0 \Rightarrow v) \wedge (w{=}1 \Rightarrow \neg v)$. Is anything wrong?

(c) Replace v with w: *nat* according to $(v \Rightarrow w{=}0) \wedge (\neg v \Rightarrow w{=}1)$. Is anything wrong?

365 Let a , b and c be boolean variables. Variables a and b are implementer's variables, and c is a user's variable for the operations

$$seta = a := \top$$
$$reseta = a := \bot$$
$$flipa = a := \neg a$$
$$setb = b := \top$$
$$resetb = b := \bot$$
$$flipb = b := \neg b$$
$$and = c := a \wedge b$$
$$or = c := a \vee b$$

This theory must be reimplemented using integer variables, with 0 for $\bot$ and all other integers for $\top$.

(a) What is the data transformer?

(b) Transform *seta* .

(c) Transform *flipa* .

(d) Transform *and* .

366 Find a data transformer to transform the program of Exercise 270(a) into the program of Exercise 270(b).

367√ (security switch) A security switch 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.

(a) Implement a security switch to correspond as directly as possible to the informal description.

(b) Transform the implementation of part (a) to obtain an efficient implementation.

368 The user's variable is boolean b . The implementer's variables are natural x and y . The operations are:

$$done = b := x{=}y{=}0$$
$$step = \textbf{if } y{>}0 \textbf{ then } y := y{-}1 \textbf{ else } (x := x{-}1.\ \textbf{var } n{:}\ nat{\cdot}\ y := n)$$

Replace the two implementer's variables x and y with a single new implementer's variable: natural z .

369 Let p be a user's boolean variable, and let m be an implementer's natural variable. The operations allow the user to assign a value n to the implementer's variable, and to test whether the implementer's variable is a prime number.

$assign\ n = m := n$
$check = p := prime\ m$

assuming *prime* is suitably defined. If *prime* is an expensive function, and the *check* operation is more frequent than the *assign* operation, we can improve the solution by making *check* less expensive even if that makes *assign* more expensive. Using data transformation, make this improvement.

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)

(a) Implement the operations in terms of bunches.
(b) Use a data transformer to replace all bunch variables with natural variables.
(c) Use a data transformer to obtain a distributed solution.

371√ 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. Here is an implementation.

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

Removing the front item from the queue takes time $p-1$ to shift all remaining items down one index. Transform the queue so that all operations are instant.

372 A binary tree can be stored as a list of nodes in breadth order. Traditionally, the root is at index 1, the node at index n has its left child at index $2\times n$ and its right child at index $2\times n+1$. Suppose the user's variable is x: X, and the implementer's variables are s: $[*X]$ and p: $nat+1$, and the operations are

$goHome = p := 1$
$goLeft = p := 2\times p$
$goRight = p := 2\times p + 1$
$goUp = p := div\ p\ 2$
$put = s := p \rightarrow x \mid s$
$get = x := s\ p$

Now suppose we decide to move the entire list down one index so that we do not waste index 0. The root is at index 0, its children are at indexes 1 and 2, and so on. Develop the necessary data transform, and use it to transform the operations.

373 (sparse array) An array A: [*[*rat]] is said to be sparse if many of its items are 0. We can represent such an array compactly as a list of triples $[i; j; x]$ of all nonzero items $A\ i\ j = x \neq 0$. Using this idea, find a data transformer and transform the programs

(a) A:= [100*[100*0]]

(b) x:= $A\ i\ j$

(c) A:= $(i;j) \rightarrow x \mid A$

374 (transformation incompleteness) The user's variable is i and the implementer's variable is j, both of type nat. The operations are:

$initialize \ = \ i' = 0 \leq j' < 3$

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

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.

(a) Show that there is no data transformer to replace j with boolean variable b so that

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

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

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.

(b) Use the data transformer $b=(j>0)$ to transform $initialize$ and $i+j=k \Rightarrow step$ where k: 0, 1, 2.

End of Theory Design and Implementation

10.8 Concurrency

375 Let x and y be natural variables. Rewrite the following program as a program that does not use $\|$.

(a) x:= $x+1$ $\|$ $\textbf{if}\ x=0\ \textbf{then}\ y := 1\ \textbf{else}\ ok$

(b) $\textbf{if}\ x>0\ \textbf{then}\ y := x-1\ \textbf{else}\ ok$ $\|$ $\textbf{if}\ x=0\ \textbf{then}\ x := y+1\ \textbf{else}\ ok$

376 If we ignore time, then

$x := 3.\ y := 4 \ = \ x := 3 \| y := 4$

Some dependent compositions could be executed in parallel if we ignore time. But the time for $P.Q$ is the sum of the times for P and Q, and that forces the execution to be sequential.

$t := t+1.\ t := t+2 \ = \ t := t+3$

Likewise some independent compositions could be executed sequentially, ignoring time. But the time for $P\|Q$ is the maximum of the times for P and Q, and that forces the execution to be parallel.

$t := t+1 \| t := t+2 \ = \ t := t+2$

Invent another form of composition, intermediate between dependent and independent composition, whose execution is sequential to the extent necessary, and parallel to the extent possible. Warning: this is a research question.

377 (disjoint composition) Independent composition $P \| Q$ requires that P and Q have no variables in common, although each can make use of the initial values of the other's variables by making a private copy. An alternative, let's say disjoint composition, is to allow both P and Q to use all the variables with no restrictions, and then to choose disjoint sets of variables v and w and define

$$P \,|v|w|\, Q \;=\; (P.\ v'{=}v) \wedge (Q.\ w'{=}w)$$

(a) Describe how $P \,|v|w|\, Q$ can be executed.

(b) Prove that if P and Q are implementable specifications, then $P \,|v|w|\, Q$ is implementable.

378 (semi-dependent composition) Independent composition $P \| Q$ requires that P and Q have no state variables in common, although each can make use of the initial values of the other's state variables by making a private copy. In this question we explore another kind of composition, let's say semi-dependent composition $P \| \| Q$. Like dependent composition, it requires P and Q to have the same state variables. Like independent composition, it can be executed by executing the processes in parallel, but each process makes its assignments to local copies of state variables. Then, when both processes are finished, the final value of a state variable is determined as follows: if both processes left it unchanged, it is unchanged; if one process changed it and the other left it unchanged, its final value is the changed one; if both processes changed it, its final value is arbitrary. This final rewriting of state variables does not require coordination or communication between the processes; each process rewrites those state variables it has changed. In the case when both processes have changed a state variable, we do not even require that the final value be one of the two changed values; the rewriting may mix the bits.

(a) Formally define semi-dependent composition, including time.

(b) What laws apply to semi-dependent composition?

(c) Under what circumstances is it unnecessary for a process to make private copies of state variables?

(d) In variables x, y, and z, without using $\| \|$, express

$$x{:=}\,z \;\|\|\; y{:=}\,z$$

(e) In variables x, y, and z, without using $\| \|$, express

$$x{:=}\,y \;\|\|\; y{:=}\,x$$

(f) In variables x, y, and z, without using $\| \|$, express

$$x{:=}\,y \;\|\|\; x{:=}\,z$$

(g) In variables x, y, and z, prove

$$x{:=}\,y \;\|\|\; x{:=}\,z \;=\; \textbf{if } x{=}y \textbf{ then } x{:=}\,z \textbf{ else if } x{=}z \textbf{ then } x{:=}\,y \textbf{ else } (x{:=}\,y \;\|\|\; x{:=}\,z)$$

(h) In boolean variables x, y and z, without using $\| \|$, express

$$x{:=}\,x{\wedge}z \;\|\|\; y{:=}\,y{\wedge}\neg z \;\|\|\; x{:=}\,x{\wedge}\neg z \;\|\|\; y{:=}\,y{\wedge}z$$

(i) Let w: 0,..4 and z: 0, 1 be variables. Without using $\| \|$, express

$$w{:=}\,2 \times \mathit{max}\ (\mathit{div}\ w\ 2)\ z \;+\; \mathit{max}\ (\mathit{mod}\ w\ 2)\ (1{-}z)$$
$$\|\|$$
$$w{:=}\,2 \times \mathit{max}\ (\mathit{div}\ w\ 2)\ (1{-}z) \;+\; \mathit{max}\ (\mathit{mod}\ w\ 2)\ z$$

379 Extend the definition of semi-dependent composition $P \| \| Q$ (Exercise 378) from variables to list items.

380 Redefine semi-dependent composition $P \| \| Q$ (Exercise 378) so that if P and Q agree on a changed value for a variable, then it has that final value, and if they disagree on a changed value for a variable, then its final value is

(a) arbitrary.

(b) either one of the two changed values.

381 We want to find the smallest number in $0,..n$ with property p. Linear search solves the problem. But evaluating p is expensive; let us say it takes time 1, and all else is free. The fastest solution is to evaluate p on all n numbers concurrently, and then find the smallest number that has the property. Write a program without concurrency for which the sequential to parallel transformation gives the desired computation.

382 Exercise 134 asks for a program to compute cumulative sums (running total). Write a program that can be transformed from sequential to parallel execution with $log\ n$ time where n is the length of the list.

383 (sieve) Given variable p: $[n*bool] := [\bot; \bot; (n–2)*\top]$, the following program is the sieve of Eratosthenes for determining if a number is prime.

for i:= 2;..*ceil* $(n^{1/2})$ **do**
  if $p\ i$ **then for** j:= i;..*ceil* (n/i) **do** p:= $(j\times i)\rightarrow\bot \mid p$
  else *ok*

(a) Show how the program can be transformed for concurrency. State your answer by drawing the execution pattern.

(b) What is the execution time, as a function of n, with maximum concurrency?

384√ (dining philosophers) 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 two chopsticks on the left and 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 with the maximum concurrency that does not lead to deadlock.

End of Concurrency

10.9 Interaction

385√ Suppose a and b are integer boundary variables, x and y are integer interactive variables, and t is an extended integer time variable. Suppose that each assignment takes time 1. Express the following using ordinary boolean operators, without using any programming notations.

$(x:= 2.\ x:= x+y.\ x:= x+y) \parallel (y:= 3.\ y:= x+y)$

386 Let a and b be boolean interactive variables. Define

$loop$ = **if** b **then** *loop* **else** *ok*

Add a time variable according to any reasonable measure, and then without using $\parallel$, express

$b:= \bot \parallel loop$

387 The Substitution Law does not work for interactive variables.

(a) Show an example of the failure of the law.

(b) Develop a new Substitution Law for interactive variables.

388√ (thermostat) Specify a thermostat for a gas burner. The thermostat operates in parallel with other processes

thermometer || *control* || *thermostat* || *burner*

The thermometer and the control are typically located together, but they are logically distinct. The inputs to the thermostat are:

- real *temperature* , which comes from the thermometer and indicates the actual temperature.
- real *desired* , which comes from the control and indicates the desired temperature.
- boolean *flame* , which comes from a flame sensor in the burner and indicates whether there is a flame.

The outputs of the thermostat are:

- boolean *gas* ; assigning it $\top$ turns the gas on and $\bot$ turns the gas off.
- boolean *spark* ; assigning it $\top$ causes sparks for the purpose of igniting the gas.

Heat is wanted when the desired temperature falls ε below the actual temperature, and not wanted when the desired temperature rises ε above the actual temperature, where ε is small enough to be unnoticeable, but large enough to prevent rapid oscillation. To obtain heat, the spark should be applied to the gas for at least 1 second to give it a chance to ignite and to allow the flame to become stable. But a safety regulation states that the gas must not remain on and unlit for more than 3 seconds. Another regulation says that when the gas is shut off, it must not be turned on again for at least 20 seconds to allow any accumulated gas to clear. And finally, the gas burner must respond to its inputs within 1 second.

389√ (grow slow) Suppose *alloc* allocates 1 unit of memory space and takes time 1 to do so. Then the following computation slowly allocates memory.

GrowSlow $\Leftarrow$ **if** $t{=}2{\times}x$ **then** (*alloc* || $x{:=}\,t$) **else** $t{:=}\,t{+}1$. *GrowSlow*

If the time is equal to $2{\times}x$, then one space is allocated, and in parallel x becomes the time stamp of the allocation; otherwise the clock ticks. The process is repeated forever. Prove that if the space is initially less than the logarithm of the time, and x is suitably initialized, then at all times the space is less than the logarithm of the time.

390 According to the definition of assignment to an interactive variable, writing to the variable takes some time during which the value of the variable is unknown. But any variables in the expression being assigned are read instantaneously at the start of the assignment. Modify the definition of assignment to an interactive variable so that

(a) writing takes place instantaneously at the end of the assignment.

(b) reading the variables in the expression being assigned takes the entire time of the assignment, just as writing does.

391 (interactive data transformation) Section 7.2 presented data transformation for boundary variables. How do we do data transformation when there are interactive variables? Warning: this is a research question.

392 (telephone) Specify the control of a simple telephone. Its inputs are those actions you can perform: picking up the phone, dialing a digit, and putting down (hanging up) the phone. Its output is a list of digits (the number dialed). The end of dialing is indicated by 5 seconds during which no further digit is dialed. If the phone is put down without waiting 5 seconds, then there is no output. But, if the phone is put down and then picked up again within 2 seconds, this is considered to be an accident, and it does not affect the output.

393 (consensus) Some parallel processes are connected in a ring. Each process has a local integer variable with an initial value. These initial values may differ, but otherwise the processes are identical. Execution of all processes must terminate in time linear in the number of processes, and in the end the values of these local variables must all be the same, and equal to one of the initial values. Write the processes.

394 Many programming languages require a variable for input, with a syntax such as **read** x . Define this form of input formally. When is it more convenient than the input described in Section 9.1? When is it less convenient?

395 Write a program to print the sequence of natural numbers, one per time unit.

396 Write a program to repeatedly print the current time, up until some given time.

397 Given a finite string S of different characters sorted in increasing order, write a program to print the strings $*(S_{0,..\leftrightarrow S})$ in the following order: shorter strings come before longer strings; strings of equal length are in string (alphabetical, lexicographic) order.

398 (T-strings) Let us call a string S: *("a", "b", "c") a T-string if no two adjacent nonempty segments are identical:

$$\neg \exists i, j, k \cdot 0 \leq i < j < k \leq \leftrightarrow S \ \wedge \ S_{i;..j} = S_{j;..k}$$

Write a program to output all T-strings in alphabetical order. (The mathematician Thue proved that there are infinitely many T-strings.)

399 (reformat) Write a program to read, reformat, and write a sequence of characters. The input includes a line-break character at arbitrary places; the output should include a line-break character just after each semicolon. Whenever the input includes two consecutive stars, or two stars separated only by line-breaks, the output should replace the two stars by an up-arrow. Other than that, the output should be identical to the input. Both input and output end with a special end-character.

400 According to the definition of **result** expression given in Subsection 5.5.0, what happens to any output that occurs in the program part of programmed data? Can input be read and used? What happens to it?

401 (Huffman code) You are given a finite set of messages, and for each message, the probability of its occurrence.

(a) Write a program to find a binary code for each message. It must be possible to unambiguously decode any sequence of 0s and 1s into a sequence of messages, and the average code length (according to message frequency) must be minimum.

(b) Write the accompanying program to produce the decoder for the codes produced in part (a).

402 (matrix multiplication) Write a program to multiply two $n \times n$ matrices that uses n^2 processes, with $2 \times n^2$ local channels, with execution time n .

403 (coin weights) You are given some coins, all of which have a standard weight except possibly for one of them, which may be lighter or heavier than the standard. You are also given a balance scale, and as many more standard coins as you need. Write a program to determine whether there is a nonstandard coin, and if so which, and whether it is light or heavy, in the minimum number of weighings.

404 How should "deterministic" and "nondeterministic" be defined in the presence of channels?

405 From the fixed-point equation

$$twos \;=\; c!\,2.\;\; t{:=}\,t{+}1.\;\; twos$$

use recursive construction to find

(a) the weakest fixed-point.

(b) a strongest implementable fixed-point.

(c) the strongest fixed-point.

406 Here are two definitions.

$$\begin{aligned} A \;=\;& \textbf{if } \surd c \wedge \surd d \textbf{ then } c? \vee d? \\ & \textbf{else if } \surd c \textbf{ then } c? \\ & \textbf{else if } \surd d \textbf{ then } d? \\ & \textbf{else if } \mathbb{T}c_{\,rc} < \mathbb{T}d_{\,rd} \textbf{ then } (t{:=}\,\mathbb{T}c_{\,rc}+1.\;\; c?) \\ & \textbf{else if } \mathbb{T}d_{\,rd} < \mathbb{T}c_{\,rc} \textbf{ then } (t{:=}\,\mathbb{T}d_{\,rd}+1.\;\; d?) \\ & \textbf{else } (t{:=}\,\mathbb{T}c_{\,rc}+1.\;\; c? \vee d?) \\ B \;=\;& \textbf{if } \surd c \wedge \surd d \textbf{ then } c? \vee d? \\ & \textbf{else if } \surd c \textbf{ then } c? \\ & \textbf{else if } \surd d \textbf{ then } d? \\ & \textbf{else } (t{:=}\,t{+}1.\;\; B) \end{aligned}$$

Letting time be an extended integer, prove $A = B$.

407 (input implementation) Let W be "wait for input on channel c and then read it".

(a)√ $$W \;=\; t{:=}\,max\; t\;(\mathbb{T}_r + 1).\;\; c?$$

Prove $W \;\Leftarrow\; \textbf{if } \surd c \textbf{ then } c? \textbf{ else } (t{:=}\,t{+}1.\;\; W)$ assuming time is an extended integer.

(b) Now let time be an extended real, redefine W appropriately, and reprove the refinement.

408 (input with timeout) As in Exercise 407, let W be "wait for input on channel c and then read it", except that if input is still not available by a deadline, an alarm should be raised.

$$W \;\Leftarrow\; \textbf{if } t \le deadline \textbf{ then if } \surd c \textbf{ then } c? \textbf{ else } (t{:=}\,t{+}1.\;\; W) \textbf{ else } alarm$$

Define W appropriately, and prove the refinement.

409 Define relation $partmerge$: $nat{\rightarrow}nat{\rightarrow}bool$ as follows:

$$\begin{aligned} & partmerge\ 0\ 0 \\ & partmerge\ (m{+}1)\ 0 \;=\; partmerge\ m\ 0 \;\wedge\; \mathbb{M}c_{\,wc+m} = \mathbb{M}a_{\,ra+m} \\ & partmerge\ 0\ (n{+}1) \;=\; partmerge\ 0\ n \;\wedge\; \mathbb{M}c_{\,wc+n} = \mathbb{M}b_{\,rb+n} \\ & partmerge\ (m{+}1)\ (n{+}1) \;=\; partmerge\ m\ (n{+}1) \;\wedge\; \mathbb{M}c_{\,wc+m+n+1} = \mathbb{M}a_{\,ra+m} \\ & \qquad\qquad\qquad\qquad \vee\; partmerge\ (m{+}1)\ n \;\wedge\; \mathbb{M}c_{\,wc+m+n+1} = \mathbb{M}b_{\,rb+n} \end{aligned}$$

Now $partmerge\ m\ n$ says that the first $m{+}n$ outputs on channel c are a merge of m inputs from channel a and n inputs from channel b. Define $merge$ as

$$merge \;=\; (a?.\;\; c!\,a) \vee (b?.\;\; c!\,b).\;\; merge$$

Prove $merge \;=\; (\forall m{\cdot}\;\exists n{\cdot}\; partmerge\ m\ n) \vee (\forall n{\cdot}\;\exists m{\cdot}\; partmerge\ m\ n)$

410 (perfect shuffle) Write a specification for a computation that repeatedly reads an input on either channel c or d. The specification says that the computation might begin with either channel, and after that it alternates.

411 (time merge) We want to repeatedly read an input on either channel c or channel d, whichever comes first, and write it on channel e. At each reading, if input is available on both channels, read either one; if it is available on just one channel, read that one; if it is available on neither channel, wait for the first one and read that one (in case of a tie, read either one).

(a)√ Write the specification formally, and then write a program.

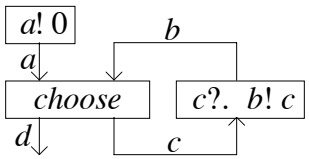
(b) Prove

$$\mathbb{T}e_{\,we} = max\ t\ (min\ (\mathbb{T}c_{\,rc})\ (\mathbb{T}d_{\,rd}) + 1)$$
$$\forall m, n \cdot \mathbb{T}e_{\,we+m+n+1} \le max\ (max\ (\mathbb{T}c_{\,rc+m})\ (\mathbb{T}d_{\,rd+n}))\ (\mathbb{T}e_{\,we+m+n}) + 1$$

412 (fairer time merge) This question is the same as the time merge (Exercise 411), but if input is available on both channels, the choice must be made the opposite way from the previous read. If, after waiting for an input, inputs arrive on both channels at the same time, the choice must be made the opposite way from the previous read.

413 In the reaction controller in Subsection 9.1.6, it is supposed that the synchronizer receives digital data from the digitizer faster than requests from the controller. Now suppose that the controller is sometimes faster than the digitizer. Modify the synchronizer so that if two or more requests arrive in a row (before new digital data arrives), the same digital data will be sent in reply to each request.

414 (Brock-Ackermann) The following picture shows a network of communicating processes.



The formal description of this network is

chan $a, b, c \cdot$ $a!\ 0 \parallel choose \parallel (c?.\ b!\ c)$

Formally define *choose*, add transit time, and state the output message and time if

(a) *choose* either reads from a and outputs a 0 on c and d, or reads from b and outputs a 1 on c and d. The choice is made freely.

(b) As in part (a), *choose* either reads from a and outputs a 0 on c and d, or reads from b and outputs a 1 on c and d. But this time the choice is not made freely; *choose* reads from the channel whose input is available first (if there's a tie, then take either one).

415√ (power series multiplication) Write a program to read from channel a an infinite sequence of coefficients $a_0\ a_1\ a_2\ a_3\ ...$ of a power series $a_0 + a_1{\times}x + a_2{\times}x^2 + a_3{\times}x^3 + ...$ and in parallel to read from channel b an infinite sequence of coefficients $b_0\ b_1\ b_2\ b_3\ ...$ of a power series $b_0 + b_1{\times}x + b_2{\times}x^2 + b_3{\times}x^3 + ...$ and in parallel to write on channel c the infinite sequence of coefficients $c_0\ c_1\ c_2\ c_3\ ...$ of the power series $c_0 + c_1{\times}x + c_2{\times}x^2 + c_3{\times}x^3 + ...$ equal to the product of the two input series. Assume that all inputs are already available; there are no input delays. Produce the outputs one per time unit.

416 (repetition) Write a program to read an infinite sequence, and after every even number of inputs, to output a boolean saying whether the second half of the input sequence is a repetition of the first half.

417 (file update) A master file of records and a transaction file of records are to be read, one record at a time, and a new file of records is to be written, one record at a time. A record consists of two text fields: a "key" field and an "info" field. The master file is kept in order of its keys, without duplicate keys, and with a final record having a sentinel key "zzzzz" guaranteed to be larger than all other keys. The transaction file is also sorted in order of its keys, with the same final sentinel key, but it may have duplicate keys. The new file is like the master file, but with changes as signified by the transaction file. If the transaction file contains a record with a key that does not appear in the master file, that record is to be added. If the transaction file contains a record with a key that does appear in the master file, that record is a change of the "info" field, unless the "info" text is the empty text, in which case it signifies record deletion. Whenever the transaction file contains a repeated key, the last record for each key determines the result.

418 (mutual exclusion) Process P is an endless repetition of a "non-critical section" N_P and a "critical section" C_P. Process Q is similar.

$$P = N_P.\ C_P.\ P$$
$$Q = N_Q.\ C_Q.\ Q$$

They are executed in parallel $(P \parallel Q)$. Specify formally that the two critical sections are never executed at the same time. Hint: You may insert into P and Q outputs on channels that are never read, but help to specify the mutual exclusion of the critical sections.

419 (synchronous communication) A synchronous communication happens when the sender is ready to send and the receiver(s) is(are) ready to receive. Those that are ready must wait for those that are not.

(a) Design a theory of synchronous communication. For each channel, you will need only one cursor, but two (or more) time scripts. An output, as well as an input, increases the time to the maximum of the time scripts for the current message.

(b) Show how it works in some examples, including a deadlock example.

(c) Show an example that is not a deadlock with asynchronous communication, but becomes a deadlock with synchronous communication.

End of Interaction

End of Exercises

11 Reference

11.0 Justifications

This section explains some of the decisions made in choosing and presenting the material in this book. It is probably not of interest to a student whose concern is to learn the material, but it may be of interest to a teacher or researcher.

11.0.0 Notation

Whenever I had to choose between a standard notation that will do and a new notation that's perfect, I chose the standard notation. For example, to express the maximum of two numbers x and y, a function max is applied: $max\ x\ y$. Since maximum is symmetric and associative, it would be better to introduce a symmetric symbol like $\uparrow$ as an infix operator: $x \uparrow y$. I always do so privately, but in this book I have chosen to keep the symbols few in number and reasonably traditional. Most people seeing $max\ x\ y$ will know what is meant without prior explanation; most people seeing $x \uparrow y$ would not. In the first edition, I used λ notation for functions, thinking that it was standard. Ten years of students convinced me that it was not standard, freeing me to use a better notation in later editions.

A precedence scheme is chosen on two criteria: to minimize the need for parentheses, and to be easily remembered. The latter is helped by sticking to tradition, by placing related symbols together, and by having as few levels as possible. The two criteria are sometimes conflicting, traditions are sometimes conflicting, and the three suggestions for helping memory are sometimes conflicting. In the end, one makes a decision and lives with it. Extra parentheses can always be used, and should be used whenever structural similarities would be obscured by the precedence scheme. For the sake of structure, it would be better to give $\wedge$ and $\vee$ the same precedence, but I have stayed with tradition. The scheme in this book has more levels than I would like. I could place $\neg$ with one-operand $-$, $\wedge$ with $\times$, $\vee$ with two-operand $+$, and $\Rightarrow$ and $\Leftarrow$ with $=$ and $\neq$. This saves four levels, but is against mathematical tradition and costs a lot of parentheses. The use of large symbols $= \quad \Leftarrow \quad \Rightarrow$ with large precedence level is a novelty; I hope it is both readable and writable. Do not judge it until you have used it awhile; it saves an enormous number of parentheses. One can immediately see generalizations of this convention to all symbols and many sizes (a slippery slope).

End of Notation

11.0.1 Basic Theories

Boolean Theory sometimes goes by other names: Boolean Algebra, Propositional Calculus, Sentential Logic. Its expressions are sometimes called “propositions” or “sentences”. Sometimes a distinction is made between “terms”, which are said to denote values, and “propositions”, which are said not to denote values but instead to be true or false. A similar distinction is made between “functions”, which apply to arguments to produce values, and “predicates”, which are instantiated to become true or false. But slowly, the subject of logic is emerging from its confused, philosophical past. I consider that propositions are just boolean expressions and treat them on a par with number expressions and expressions of other types. I consider that predicates are just boolean functions. I use the same equal sign for booleans as for numbers, characters, sets, and functions. Perhaps in the future we won't feel the need to imagine abstract objects for expressions to denote; we will justify them by their practical applications. We will explain our formalisms by the rules for their use, not by their philosophy.

Why bother with “antiaxioms” and “antitheorems”? They are not traditional (in fact, I made up the words). As stated in Chapter 1, thanks to the negation operator and the Consistency Rule, we don't need to bother with them. Instead of saying that *expression* is an antitheorem, we can say that $\neg$*expression* is a theorem. Why bother with $\bot$? We could instead write $\neg\top$. One reason is just that it is shorter to say “antitheorem” than to say “negation of a theorem”. Another reason is to help make clear the important difference between “disprovable” and “not provable”. Another reason is that some logics do not use the negation operator and the Consistency Rule. The logic in this book is “classical logic”; “constructive logic” omits the Completion Rule; “evaluation logic” omits both the Consistency Rule and the Completion Rule.

Some books present proof rules (and axioms) with the aid of a formal metanotation. In this book, there is no formal metalanguage; the metalanguage is English. A formal metalanguage may be considered helpful (though it is not necessary) for the presentation and comparison of a variety of competing formalisms, and for proving theorems about formalisms. But in this book, only one formalism is presented. The burden of learning another formalism first, for the purpose of presenting the main formalism, is unnecessary. A formal metanotation [/] for substitution would allow me to write the function application rule as

$\langle v \rightarrow b \rangle\, a = b[a/v]$

but then I would have to explain that $b[a/v]$ means “substitute a for v in b”. I may as well say directly

$\langle v \rightarrow b \rangle\, a =$ (substitute a for v in b)

A proof syntax (formalizing the “hints”) would be necessary if we were using an automated prover, but in this book it is unnecessary and I have not introduced one.

Some authors may distinguish “axiom” from “axiom schema”, the latter having variables which can be instantiated to produce axioms; I have used the term “axiom” for both. I have also used the term “law” as a synonym for “theorem” (I would prefer to reduce my vocabulary, but both words are well established). Other books may distinguish them by the presence or absence of variables, or they may use “law” to mean “we would like it to be a theorem but we haven't yet designed an appropriate theory”.

I have taken a few liberties with the names of some axioms and laws. What I have called “transparency” is often called “substitution of equals for equals”, which is longer and doesn't quite make sense. Each of my Laws of Portation is historically two laws, one an implication in one direction, and the other an implication in the other direction. One was called “Importation”, and the other “Exportation”, but I can never remember which was which.

End of Basic Theories

11.0.2 Basic Data Structures

Why bother with bunches? Don't sets work just as well? Aren't bunches really just sets but using a peculiar notation and terminology? The answer is no, but let's take it slowly. Suppose we just present sets. We want to be able to write $\{1, 3, 7\}$ and similar expressions, and we might describe these set expressions with a little grammar like this:

set = "{" *contents* "}"
contents = *number*
| *set*
| *contents* "," *contents*

We will want to say that the order of elements in a set is irrelevant so that $\{1, 2\} = \{2, 1\}$; the best way to say it is formally: $A,B = B,A$ (comma is symmetric, or commutative). Next, we want to say

that repetitions of elements in a set are irrelevant so that {3, 3} = {3} ; the best way to say that is *A*,*A* = *A* (comma is idempotent). What we are doing here is inventing bunches, but calling them "contents" of a set. And note that the grammar is equating bunches; the string catenations (denoted here by juxtaposition) distribute over the elements of their operands, and the alternations (denoted here by vertical bars) are bunch unions.

When a child first learns about sets, there is often an initial hurdle: that a set with one element is not the same as the element. How much easier it would be if a set were presented as packaging: a bag with an apple in it is obviously not the same as the apple. Just as {2} and 2 differ, so {2,7} and 2,7 differ. Bunch Theory tells us about aggregation; Set Theory tells us about packaging. The two are independent.

We could define sets without relying on bunches (as has been done for many years), and we could use sets wherever I have used bunches. In that sense, bunches are unnecessary. Similarly we could define lists without relying on sets (as I did in this book), and we could always use lists in place of sets. In that sense, sets are unnecessary. But sets are a beautiful data structure that introduces one idea (packaging), and I prefer to keep them. Similarly bunches are a beautiful data structure that introduces one idea (aggregation), and I prefer to keep them. I always prefer to use the simplest structure that is adequate for its purpose.

The subject of functional programming has suffered from an inability to express nondeterminism conveniently. To say something about a value, but not pin it down completely, one can express the set of possible values. Unfortunately, sets do not reduce properly to the deterministic case; in this context it is again a problem that a set containing one element is not equal to the element. What is wanted is bunches. One can always regard a bunch as a "nondeterministic value".

Bunches have also been used in this book as a "type theory". Surely it is discouraging to others, as it is to me, to see type theory duplicating all the operators of its value space: for each operation on values, there is a corresponding operation on type spaces. By using bunches, this duplication is eliminated.

Many mathematicians consider that curly brackets and commas are just syntax, and syntax is annoying and unimportant, though necessary. I have treated them as operators, with algebraic properties (in Section 2.1 on Set Theory, we see that curly brackets have an inverse). This continues a very long, historical trend. For example, = was at first just a syntax for the informal statement that two things are (in some way) the same, but now it is a formal operator with algebraic properties.

In many papers there is a little apology as the author explains that the notation for catenation of lists will be abused by sometimes catenating a list and an item. Or perhaps there are three catenation notations: one to catenate two lists, one to prepend an item to a list, and one to append an item to a list. The poor author has to fight with unwanted packaging provided by lists in order to get the sequencing. I offer these authors strings: sequencing without packaging. (Of course, they can be packaged into lists whenever wanted. I am not taking away lists.)

End of Basic Data Structures

11.0.3 Function Theory

I have used the words "local" and "nonlocal" where others might use the words "bound" and "free", or "local" and "global", or "hidden" and "visible", or "private" and "public". The tradition in logic, which I have not followed, is to begin with all possible variables (infinitely many of them) already "existing". The function notation $\langle\ \rangle$ is said to "bind" variables, and any variable that is not bound remains "free". For example, $\langle x{:}\ int \rightarrow x+y \rangle$ has bound variable x, free variable y, and infinitely many other free variables. In this book, variables do not automatically "exist"; they are introduced (rather than bound) either formally using the function notation, or informally by saying in English what they are.

The quantifier formed from *max* is called *MAX* even though its result may not be any result of the function it is applied to; the name "least upper bound" is traditional. Similarly for *MIN*, which is traditionally called "greatest lower bound".

I have ignored the traditional question of the "existence" of limits; in cases where traditionally a limit does not "exist", the Limit Law does not tell us exactly what the limit is, but it might still tell us something useful.

End of Function Theory

11.0.4 Program Theory

Assignment could have been defined as

$$x{:=}\ e \ =\ \textit{defined}\ \text{"}e\text{"} \wedge e{:}\ T \Rightarrow x'{=}e \wedge y'{=}y \wedge \ldots$$

where *defined* rules out expressions like $1/0$, and T is the type of variable x. I left out *defined* because a complete definition of it is impossible, a reasonably complete definition is as complicated as all of program theory, and it serves no purpose. The antecedent $e{:}\ T$ would be useful, making the assignment $n{:=}\ n{-}1$ implementable when n is a natural variable. But its benefit is not worth its trouble, since the same check is made at every dependent composition. Even worse, we would lose the Substitution Law; we want $(n{:=}\ {-}1.\ n{\geq}0)$ to be $\bot$.

Since the design of Algol-60, sequential execution has often been represented by a semi-colon. The semi-colon is unavailable to me for this purpose because I used it for string catenation. Dependent composition is a kind of product, so I hope a period will be an acceptable symbol. I considered switching the two, using semi-colon for dependent composition and a period for string catenation, but the latter did not work well.

In English, the word "precondition" means "something that is necessary beforehand". In many programming books, the word "precondition" is used to mean "something that is sufficient beforehand". In those books, "weakest precondition" means "necessary and sufficient precondition", which I have called "exact precondition".

In the earliest and still best-known theory of programming, we specify that variable x is to be increased as follows:

$$\{x = X\}\ S\ \{x > X\}$$

We are supposed to know that x is a state variable, that X is a local variable to this specification whose purpose is to relate the initial and final value of x, and that S is also local to the specification and is a place-holder for a program. Neither X nor S will appear in a program that refines this specification. Formally, X and S are quantified as follows:

$$\S S{\cdot}\ \forall X{\cdot}\ \{x = X\}\ S\ \{x > X\}$$

In the theory of weakest preconditions, the equivalent specification looks similar:

$$\S S\cdot \forall X\cdot x{=}X \Rightarrow wp\ S\ (x{>}X)$$

There are two problems with these notations. One is that they do not provide any way of referring to both the prestate and the poststate, hence the introduction of X. This is solved in the Vienna Development Method, in which the same specification is

$$\S S\cdot \{\mathsf{T}\}\ S\ \{x' > x\}$$

The other problem is that the programming language and specification language are disjoint, hence the introduction of S. In my theory, the programming language is a sublanguage of the specification language. The specification that x is to be increased is

$$x' > x$$

The same single-expression double-state specifications are used in Z, but refinement is rather complicated. In Z, P is refined by S if and only if

$$\forall\sigma\cdot (\exists\sigma'\cdot P) \Rightarrow (\exists\sigma'\cdot S) \wedge (\forall\sigma'\cdot P \Leftarrow S)$$

In the early theory, $\S S\cdot \{P\}\ S\ \{Q\}$ is refined by $\S S\cdot \{R\}\ S\ \{U\}$ if and only if

$$\forall\sigma\cdot P \ \Rightarrow\ R \wedge (Q \Leftarrow U)$$

In my theory, P is refined by S if and only if

$$\forall\sigma, \sigma'\cdot P \Leftarrow S$$

Since refinement is what we must prove when programming, it is best to make refinement as simple as possible.

One might suppose that any type of mathematical expression can be used as a specification: whatever works. A specification of something, whether cars or computations, distinguishes those things that satisfy it from those that don't. Observation of something provides values for certain variables, and on the basis of those values we must be able to determine whether the something satisfies the specification. Thus we have a specification, some values for variables, and two possible outcomes. That is exactly the job of a boolean expression: a specification (of anything) really is a boolean expression. If instead we use a pair of predicates, or a function from predicates to predicates, or anything else, we make our specifications in an indirect way, and we make the task of determining satisfaction more difficult.

One might suppose that any boolean expression can be used to specify any computer behavior: whatever correspondence works. In Z, the expression T is used to specify (describe) terminating computations, and $\bot$ is used to specify (describe) nonterminating computations. The reasoning is something like this: $\bot$ is the specification for which there is no satisfactory final state; an infinite computation is behavior for which there is no final state; hence $\bot$ represents infinite computation. Although we cannot observe a "final" state of an infinite computation, we can observe, simply by waiting 10 time units, that it satisfies $t' > t{+}10$, and it does not satisfy $t' \le t{+}10$. Thus it ought to satisfy any specification implied by $t' > t{+}10$, including T, and it ought not to satisfy any specification that implies $t' \le t{+}10$, including $\bot$. Since $\bot$ is not true of anything, it does not describe anything. A specification is a description, and $\bot$ is not satisfiable, not even by nonterminating computations. Since T is true of everything, it describes everything, even nonterminating computations. To say that P refines Q is to say that all behavior satisfying P also satisfies Q, which is just implication. The correspondence between specifications and computer behavior is not arbitrary.

As pointed out in Chapter 4, specifications such as $x'{=}2 \wedge t'{=}\infty$ that talk about the "final" values of variables at time infinity are strange. I could change the theory to prevent any mention of results at time infinity, but I do not for two reasons: it would make the theory more complicated, and I need to distinguish among infinite loops when I introduce interactions (Chapter 9).

End of Program Theory

11.0.5 Programming Language

The form of variable declaration given in Chapter 5 assigns the new local variable an arbitrary value of its type. Thus, for example, if y and z are integer variables, then

var x: $nat \cdot y := x \;=\; y'$: $nat \;\wedge\; z'=z$

For ease of implementation and speed of execution, this is much better than initialization with "the undefined value". For error detection, it is no worse, assuming that we prove all our refinements. Furthermore, there are circumstances in which arbitrary initialization is exactly what's wanted (see Exercise 270 (majority vote)). However, if we do not prove all our refinements, initialization with *undefined* provides a measure of protection. If we allow the generic operators (= , ≠ , **if then else**) to apply to *undefined* , then we can prove trivialities like *undefined* = *undefined* . If not, then we can prove nothing at all about *undefined* . Some programming languages seek to eliminate the error of using an uninitialized variable by initializing each variable to a standard value of its type. Such languages achieve the worst of all worlds: they are not as efficient as arbitrary initialization; and they eliminate only the error detection, not the error.

An alternative way to define variable declaration is

var x: $T \;=\; x'$: $T \;\wedge\; ok$

which starts the scope of x , and

end $x \;=\; ok$

which ends the scope of x . In each of these programs, ok maintains the other variables. This kind of declaration does not require scopes to be nested; they can be overlapped.

The most widely known and used rule for **while**-loops is the Method of Invariants and Variants. Let I be a precondition (called the "invariant") and let I' be the corresponding postcondition. Let v be an integer expression (called the "variant" or "bound function") and let v' be the corresponding expression with primed variables. The Rule of Invariants and Variants says:

$I \Rightarrow I' \wedge \neg b' \;\Leftarrow\;$ **while** b **do** $I \wedge b \;\Rightarrow\; I' \wedge 0 \le v' < v$

The rule says, very roughly, that if the body of the loop maintains the invariant and decreases the variant but not below zero, then the loop maintains the invariant and negates the loop condition. For example, to prove

$s' = s + \Sigma\, L\,[n;..\#L] \;\Leftarrow\;$ **while** $n \neq \#L$ **do** $(s := s + Ln.\;\; n := n+1)$

we must invent an invariant

$s + \Sigma\, L\,[n;..\#L] = \Sigma L$

and a variant

$\#L - n$

and prove both

$$s' = s + \Sigma\, L\,[n;..\#L]$$
$$\Leftarrow \quad s + \Sigma\, L\,[n;..\#L] = \Sigma L \;\Rightarrow\; s' + \Sigma\, L\,[n';..\#L] = \Sigma L \;\wedge\; n'=\#L$$

and

$$s + \Sigma\, L\,[n;..\#L] = \Sigma L \;\wedge\; n \neq \#L \;\Rightarrow\; s' + \Sigma\, L\,[n';..\#L] = \Sigma L \;\wedge\; 0 \le \#L - n' < \#L - n$$
$$\Leftarrow \quad s := s + Ln.\;\; n := n+1$$

The proof method given in Chapter 5 is easier and more information (time) is obtained. Sometimes the Method of Invariants and Variants requires the introduction of extra constants (mathematical variables) not required by the proof method in Chapter 5. For example, to add 1 to each item in list L requires introducing list constant M to stand for the initial value of L .

Probability Theory would be simpler if all real numbers were probabilities, instead of just the reals in the closed interval from 0 to 1 , in which case I would add the axioms $\top = \infty$ and $\bot = -\infty$; but it is not my purpose in this book to invent a better probability theory. For probabilistic

programming, my first approach was to reinterpret the types of variables as probability distributions expressed as functions. If x was a variable of type T, it becomes a variable of type $T \rightarrow prob$ such that $\Sigma x = \Sigma x' = 1$. All operators then need to be extended to distributions expressed as functions. Although this approach works, it was too low-level; a distribution expressed as a function tells us about the probability of its variables by their positions in an argument list, rather than by their names.

The subject of programming has often been mistaken for the learning of a large number of programming language "features". This mistake has been made of both imperative and functional programming. Of course, each fancy operator provided in a programming language makes the solution of some problems easy. In functional programming, an operator called "fold" or "reduce" is often presented; it is a useful generalization of some quantifiers. Its symbol might be / and it takes as left operand a two-operand operator and as right operand a list. The list summation problem is solved as $+/L$. The search problem could similarly be solved by the use of an appropriate search operator, and it would be a most useful exercise to design and implement such an operator. This exercise cannot be undertaken by someone whose only programming ability is to find an already implemented operator and apply it. The purpose of this book is to teach the necessary programming skills.

As our examples illustrate, functional programming and imperative programming are essentially the same: the same problem in the two styles requires the same steps in its solution. They have been thought to be different for the following reasons: imperative programmers adhere to clumsy loop notations, complicating proofs; functional programmers adhere to equality, rather than refinement, making nondeterminism difficult.

End of Programming Language

11.0.6 Recursive Definition

The combination of construction and induction is so useful that it has a name (generation) and a notation (::=). To keep terminology and notation to a minimum, I have not used them.

Recursive construction has always been done by taking the limit of a sequence of approximations. My innovation is to substitute ∞ for the index in the sequence; this is a lot easier than finding a limit. Substituting ∞ is not guaranteed to produce the desired fixed-point, but neither is finding the limit. Substituting ∞ works well except in examples contrived to show its limitation.

End of Recursive Definition

11.0.7 Theory Design and Implementation

I used the term "data transformation" instead of the term "data refinement" used by others. I don't see any reason to consider one space more "abstract" and another more "concrete". What I call a "data transformer" is sometimes called "abstraction relation", "linking invariant", "gluing relation", "retrieve function", or "data invariant".

The incompleteness of data transformation is demonstrated with an example carefully crafted to show the incompleteness, not one that would ever arise in practice. I prefer to stay with the simple rule that is adequate for all transformations that will ever arise in any problem other than a demonstration of theoretical incompleteness, rather than to switch to a more complicated rule, or combination of rules, that are complete. To regain completeness, all we need is the normal mathematical practice of introducing local variables. Variables for this purpose have been called

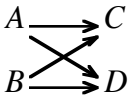
"bound variables", "logical constants", "specification variables", "ghost variables", "abstract variables", and "prophesy variables", by different authors.

End of Theory Design and Implementation

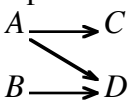
11.0.8 Concurrency

In FORTRAN (prior to 1977) we could have a sequential composition of **if**-statements, but we could not have an **if**-statement containing a sequential composition. In ALGOL the syntax was fully recursive; sequential and conditional compositions could be nested, each within the other. Did we learn a lesson? Apparently we did not learn a very general one: we now seem happy to have a parallel composition of sequential compositions, but very reluctant to have a sequential composition of parallel compositions. So in currently popular languages, a parallel composition can occur only as the outermost construct.

As we saw in Chapter 8, the execution pattern



can be expressed as $((A \parallel B).\ (C \parallel D))$ without any synchronization primitives. But the pattern



cannot be expressed using only parallel and sequential composition. This pattern occurs in the buffer program.

In the first edition of this book, parallel composition was defined for processes having the same state space (semi-dependent composition). That definition was more complicated than the present one (see Exercise 378), but in theory, it eliminated the need to partition the variables. In practice, however, the variables were always partitioned, so in the present edition the simpler definition (independent composition) is used.

End of Concurrency

11.0.9 Interaction

In the formula for implementability, there is no conjunct $r' \le w'$ saying that the read cursor must not get ahead of the write cursor. In Subsection 9.1.8 on deadlock we see that it can indeed happen. Of course, it takes infinite time to do so. In the deadlock examples, we can prove that the time is infinite. But there is a mild weakness in the theory. Consider this example.

$$\begin{aligned} & \textbf{chan } c\cdot\ t := max\ t\ (\mathbb{T}_r + 1).\ c? \\ = \ & \exists \mathbb{M}, \mathbb{T}, r, r', w, w' \cdot\ t' = max\ t\ (\mathbb{T}_0 + 1)\ \wedge\ r'=1\ \wedge\ w'=0 \\ = \ & t' \ge t \end{aligned}$$

We might like to prove $t'=\infty$. To get this answer, we must strengthen the definition of local channel declaration by adding the conjunct $\mathbb{T}_{w'} \ge t'$. I prefer the simpler, weaker theory.

End of Interaction

We could talk about a structure of channels, and about indexed processes. We could talk about a parallel **for**-loop. There is always something more to say, but we have to stop somewhere.

End of Justifications

11.1 Sources

Ideas do not come out of nowhere. They are the result of one's education, one's culture, and one's interactions with acquaintances. I would like to acknowledge all those people who have influenced me and enabled me to write this book. I will probably fail to mention people who have influenced me indirectly, even though the influence may be strong. I may fail to thank people who gave me good ideas on a bad day, when I was not ready to understand. I will fail to give credit to people who worked independently, whose ideas may be the same as or better than those that happened to reach my eyes and ears. To all such people, I apologize. I do not believe anyone can really take credit for an idea. Ideally, our research should be done for the good of everyone, perhaps also for the pleasure of it, but not for the personal glory. Still, it is disappointing to be missed. Here then is the best accounting of my sources that I can provide.

The early work in this subject is due to Alan Turing (1949), Peter Naur (1966), Robert Floyd (1967), Tony Hoare (1969), Rod Burstall (1969), and Dana Scott and Christopher Strachey (1970). (See the Bibliography, which follows.) My own introduction to the subject was a book by Edsger Dijkstra (1976); after reading it I took my first steps toward formalizing refinement (1976). Further steps in that same direction were taken by Ralph Back (1978), though I did not learn of them until 1984. The first textbooks on the subject began to appear, including one by me (1984). That work was based on Dijkstra's weakest precondition predicate transformer, and work continues today on that same basis. I highly recommend the book *Refinement Calculus* by Ralph Back and Joachim vonWright (1998).

In the meantime, Tony Hoare (1978, 1981) was developing communicating sequential processes. During a term at Oxford in 1981 I realized that they could be described as predicates, and published a predicate model (1981, 1983). It soon became apparent that the same sort of description, a single boolean expression, could be used for any kind of computation, and indeed for anything else; in retrospect, it should have been obvious from the start. The result was a series of papers (1984, 1986, 1988, 1989, 1990, 1994, 1998, 1999, 2004) leading to the present book.

The importance of format in expressions and proofs was made clear to me by Netty van Gasteren (1990). The symbols $¢$ and $\$$ for bunch and set cardinality were suggested by Chris Lengauer. The word "conflation" was suggested by Doug McIlroy. The value of indexing from 0 was taught to me by Edsger Dijkstra. Joe Morris and Alex Bunkenburg (2001) found and fixed a problem with bunch theory. The word "apposition" and the idea to which it applies come from Lambert Meertens (1986). Peter Kanareitsev helped with higher-order functions. Alan Rosenthal suggested that I stop worrying about when limits "exist", and just write the axioms describing them; I hope that removes the last vestige of Platonism from the mathematics, though some remains in the English. My Refinement by Parts law was made more general by Theo Norvell. I learned the use of a timing variable from Chris Lengauer (1981), who credits Mary Shaw; we were using weakest preconditions then, so our time variables ran down instead of up. The recursive measure of time is inspired by the work of Paul Caspi, Nicolas Halbwachs, Daniel Pilaud, and John Plaice (1987); in their language LUSTRE, each iteration of a loop takes time 1, and all else is free. I learned to discount termination by itself, with no time bound, in discussions with Andrew Malton, and from an example of Hendrik Boom (1982). I was told the logarithmic solution to the Fibonacci number problem by Wlad Turski, who learned it while visiting the University of Guelph. My incorrect version of local variable declaration was corrected by Andrew Malton. Local variable suspension is adapted from Carroll Morgan (1990). The **for**-loop rule was influenced by Victor Kwan and Emil Sekerinski. The backtracking implementation of unimplementable specifications is an adaptation of a technique due to Greg Nelson (1989) for implementing angelic nondeterminism.

Carroll Morgan and Annabelle McIver (1996) suggested probabilities as observable quantities, and Exercise 284 (Mr.Bean's socks) comes from them. The use of bunches for nondeterminism in functional programming and for function refinement is joint work with Theo Norvell (1992). Theo also added the timing to the recursive definition of **while**-loops (1997). The style of data-type theories (data-stack, data-queue, data-tree) comes from John Guttag and Jim Horning (1978). The implementation of data-trees was influenced by Tony Hoare (1975). Program-tree theory went through successive versions due to Theo Norvell, Yannis Kassios, and Peter Kanareitsev. I learned about data transformation from He Jifeng and Carroll Morgan, based on earlier work by Tony Hoare (1972); the formulation here is my own, but I checked it for equivalence with those in Wei Chen and Jan Tijmen Udding (1989). Theo Norvell provided the criterion for data transformers. The second data transformation example (take a number) is adapted from a resource allocation example of Carroll Morgan (1990). The final data transformation example showing incompleteness was invented by Paul Gardiner and Carroll Morgan (1993). For an encyclopedic treatment of data transformers, see the book by Willem-Paul deRoever and Kai Engelhardt (1998). I published various formulations of independent (parallel) composition (1981, 1984, 1990, 1994); the one in the first edition of this book is due to Theo Norvell and appears in this edition as Exercise 378 (semi-dependent composition), and is used in recent work by Hoare and He (1998); for this edition I was persuaded by Leslie Lamport to return to my earlier (1984, 1990) version: simple conjunction. Section 8.1 on sequential to parallel transformation is joint work with Chris Lengauer (1981); he has since made great advances in the automatic production of highly parallel, systolic computations from ordinary sequential, imperative programs. The thermostat example is a simplification and adaptation of a similar example due to Anders Ravn, Erling Sørensen, and Hans Rischel (1990). The form of communication was influenced by Gilles Kahn (1974). Time scripts were suggested by Theo Norvell. The input check is an invention of Alain Martin (1985), which he called the “probe”. Monitors were invented by Per Brinch Hansen (1973) and Tony Hoare (1974). The power series multiplication is from Doug McIlroy (1990), who credits Gilles Kahn. Many of the exercises were given to me by Wim Feijen for my earlier book (1984); they were developed by Edsger Dijkstra, Wim Feijen, Netty van Gasteren, and Martin Rem for examinations at the Technical University of Eindhoven; they have since appeared in a book by Edsger Dijkstra and Wim Feijen (1988). Some exercises come from a series of journal articles by Martin Rem (1983,..1991). Other exercises were taken from a great variety of sources too numerous to mention.

End of Sources

11.2 Bibliography

R.-J.R.Back: “on the Correctness of Refinement Steps in Program Development”, University of Helsinki, Department of Computer Science, Report A-1978-4, 1978

R.-J.R.Back: “a Calculus of Refinement for Program Derivations”, *Acta Informatica*, volume 25, pages 593,..625, 1988

R.-J.R.Back, J.vonWright: *Refinement Calculus: a Systematic Introduction*, Springer, 1998

H.J.Boom: “a Weaker Precondition for Loops”, *ACM Transactions on Programming Languages and Systems*, volume 4, number 4, pages 668,..678, 1982

P.BrinchHansen: “Concurrent Programming Concepts”, *ACM Computing Surveys*, volume 5, pages 223,..246, 1973 December

R.Burstall: “Proving Properties of Programs by Structural Induction”, University of Edinburgh, Report 17 DMIP, 1968; also *Computer Journal*, volume 12, number 1, pages 41,..49, 1969

P.Caspi, N.Halbwachs, D.Pilaud, J.A.Plaice: “LUSTRE: a Declarative Language for Programming Synchronous Systems”, *fourteenth annual ACM Symposium on Principles of Programming Languages*, pages 178,..189, Munich, 1987

K.M.Chandy, J.Misra: *Parallel Program Design: a Foundation*, Addison-Wesley, 1988

W.Chen, J.T.Udding: “Toward a Calculus of Data Refinement”, J.L.A.van de Snepscheut (editor): *Mathematics of Program Construction*, Springer, Lecture Notes in Computer Science, volume 375, pages 197,..219, 1989

E.W.Dijkstra: “Guarded Commands, Nondeterminacy, and Formal Derivation of Programs”, *Communications ACM*, volume 18, number 8, pages 453,..458, 1975 August

E.W.Dijkstra: *a Discipline of Programming*, Prentice-Hall, 1976

E.W.Dijkstra, W.H.J.Feijen: *a Method of Programming*, Addison-Wesley, 1988

R.W.Floyd: “Assigning Meanings to Programs”, *Proceedings of the American Society, Symposium on Applied Mathematics*, volume 19, pages 19,..32, 1967

P.H.B.Gardiner, C.C.Morgan: “a Single Complete Rule for Data Refinement”, *Formal Aspects of Computing*, volume 5, number 4, pages 367,..383, 1993

A.J.M.vanGasteren: “on the Shape of Mathematical Arguments”, Springer-Verlag Lecture Notes in Computer Science, 1990

J.V.Guttag, J.J.Horning: “the Algebraic Specification of Abstract Data Types”, *Acta Informatica*, volume 10, pages 27,..53, 1978

E.C.R.Hehner: "**do** considered **od**: a Contribution to the Programming Calculus", University of Toronto, Technical Report CSRG-75, 1976 November; also *Acta Informatica*, volume 11, pages 287,..305, 1979

E.C.R.Hehner: "Bunch Theory: a Simple Set Theory for Computer Science", University of Toronto, Technical Report CSRG-102, 1979 July; also *Information Processing Letters*, volume 12, number 1, pages 26,..31, 1981 February

E.C.R.Hehner, C.A.R.Hoare: "a More Complete Model of Communicating Processes", University of Toronto, Technical Report CSRG-134, 1981 September; also *Theoretical Computer Science*, volume 26, numbers 1 and 2, pages 105,..121, 1983 September

E.C.R.Hehner: "Predicative Programming", *Communications ACM*, volume 27, number 2, pages 134,..152, 1984 February

E.C.R.Hehner: *the Logic of Programming*, Prentice-Hall International, 1984

E.C.R.Hehner, L.E.Gupta, A.J.Malton: "Predicative Methodology", *Acta Informatica*, volume 23, number 5, pages 487,..506, 1986

E.C.R.Hehner, A.J.Malton: "Termination Conventions and Comparative Semantics", *Acta Informatica*, volume 25, number 1, pages 1,..15, 1988 January

E.C.R.Hehner: "Termination is Timing", Conference on Mathematics of Program Construction, The Netherlands, Enschede, 1989 June; also J.L.A.van de Snepscheut (editor): *Mathematics of Program Construction*, Springer-Verlag, Lecture Notes in Computer Science volume 375, pages 36,..48, 1989

E.C.R.Hehner: "a Practical Theory of Programming", *Science of Computer Programming*, volume 14, numbers 2 and 3, pages 133,..159, 1990

E.C.R.Hehner: "Abstractions of Time", *a Classical Mind,* chapter 12, Prentice-Hall, 1994

E.C.R.Hehner: "Formalization of Time and Space", *Formal Aspects of Computing*, volume 10, pages 290,..307, 1998

E.C.R.Hehner, A.M.Gravell: "Refinement Semantics and Loop Rules", FM'99 World Congress on Formal Methods, pages 20,..25, Toulouse France, 1999 September

E.C.R.Hehner: "Specifications, Programs, and Total Correctness", *Science of Computer Programming* volume 34, pages 191,..206, 1999

E.C.R.Hehner: "Probabilistic Predicative Programming", Conference on Mathematics of Program Construction, Scotland, Stirling, 2004 July 12,..15, and Springer Lecture Notes in Computer Science, D.M.Kozen (editor), volume 3125, pages 169,..186, 2004

C.A.R.Hoare: "an Axiomatic Basis for Computer Programming", *Communications ACM*, volume 12, number 10, pages 576,..581, 583, 1969 October

C.A.R.Hoare: “Proof of Correctness of Data Representations”, *Acta Informatica*, volume 1, number 4, pages 271,..282, 1972

C.A.R.Hoare: “Monitors: an Operating System Structuring Concept”, *Communications ACM*, volume 17, number 10, pages 549,..558, 1974 October

C.A.R.Hoare: “Recursive Data Structures”, *International Journal of Computer and Information Sciences*, volume 4, number 2, pages 105,..133, 1975 June

C.A.R.Hoare: “Communicating Sequential Processes”, *Communications ACM*, volume 21, number 8, pages 666,..678, 1978 August

C.A.R.Hoare: “a Calculus of Total Correctness for Communicating Processes”, *Science of Computer Programming*, volume 1, numbers 1 and 2, pages 49,..73, 1981 October

C.A.R.Hoare: “Programs are Predicates”, in C.A.R.Hoare, J.C.Shepherdson (editors): *Mathematical Logic and Programming Languages*, Prentice-Hall Intenational, pages 141,..155, 1985

C.A.R.Hoare, I.J.Hayes, J.He, C.C.Morgan, A.W.Roscoe, J.W.Sanders, I.H.Sørensen, J.M.Spivey, B.A.Sufrin: “the Laws of Programming”, *Communications ACM*, volume 30, number 8, pages 672,..688, 1987 August

C.A.R.Hoare, J.He: *Unifying Theories of Programming*, Prentice-Hall, 1998

C.B.Jones: *Software Development: a Rigorous Approach*, Prentice-Hall International, 1980

C.B.Jones: *Systematic Software Development using VDM*, Prentice-Hall International, 1990

G.Kahn: “the Semantics of a Simple Language for Parallel Programming”, *Information Processing 74*, North-Holland, Proceeding of IFIP Congress, 1974

C.Lengauer, E.C.R.Hehner: “a Methodology for Programming with Concurrency”, CONPAR 81, Nürnberg, 1981 June 10,..13; also Springer-Verlag, Lecture Notes in Computer Science volume 111, pages 259,..271, 1981 June; also *Science of Computer Programming*, volume 2, pages 1,..53 , 1982

A.J.Martin: “the Probe: an Addition to Communication Primitives”, *Information Processing Letters*, volume 20, number 3, pages 125,..131, 1985

J.McCarthy: “a Basis for a Mathematical Theory of Computation”, *Proceedings of the Western Joint Computer Conference*, pages 225,..239, Los Angeles, 1961 May; also Computer Programming and Formal Systems, North-Holland, pages 33,..71, 1963

M.D.McIlroy: “Squinting at Power Series”, *Software Practice and Experience*, volume 20, number 7, pages 661,..684, 1990 July

L.G.L.T.Meertens: “Algorithmics — towards Programming as a Mathematical Activity”, Proceedings of CWI Symposium on Mathematics and Computer Science, North-Holland, *CWI Monographs*, volume 1, pages 289,..335, 1986

C.C.Morgan: “the Specification Statement”, *ACM Transactions on Programming Languages and Systems*, volume 10, number 3, pages 403,..420, 1988 July

C.C.Morgan: *Programming from Specifications*, Prentice-Hall International, 1990

C.C.Morgan, A.K.McIver, K.Seidel, J.W.Sanders: “Probabilistic Predicate Transformers”, *ACM Transactions on Programming Languages and Systems*, volume 18, number 3, pages 325,..354, 1996 May

J.M.Morris: “a Theoretical Basis for Stepwise Refinement and the Programming Calculus”, *Science of Computer Programming*, volume 9, pages 287,..307, 1987

J.M.Morris, A.Bunkenburg: “a Theory of Bunches”, *Acta Informatica*, volume 37, number 8, pages 541,..563, 2001 May

P.Naur: “Proof of Algorithms by General Snapshots”, *BIT*, volume 6, number 4, pages 310,..317, 1966

G.Nelson: “a Generalization of Dijkstra's Calculus”, *ACM Transactions on Programming Languages and Systems*, volume 11, number 4, pages 517,..562, 1989 October

T.S.Norvell: “Predicative Semantics of Loops”, *Algorithmic Languages and Calculi*, Chapman-Hall, 1997

T.S.Norvell, E.C.R.Hehner: “Logical Specifications for Functional Programs”, International Conference on Mathematics of Program Construction, Oxford, 1992 June

A.P.Ravn, E.V.Sørensen, H.Rischel: “Control Program for a Gas Burner”, Technical University of Denmark, Department of Computer Scence, 1990 March

M.Rem: “Small Programming Exercises”, articles in *Science of Computer Programming*, 1983,..1991

W.-P.deRoever, K.Engelhardt: *Data Refinement: Model-Oriented Proof Methods and their Comparisons*, tracts in Theoretical Computer Science volume 47, Cambridge University Press, 1998

D.S.Scott, C.Strachey: “Outline of a Mathematical Theory of Computation”, technical report PRG-2, Oxford University, 1970; also *Proceedings of the fourth annual Princeton Conference on Information Sciences and Systems*, pages 169,..177, 1970

K.Seidel, C.Morgan, A.K.McIver: “an Introduction to Probabilistic Predicate Transformers”, technical report PRG-TR-6-96, Oxford University, 1996

J.M.Spivey: *the Z Notation – a Reference Manual*, Prentice-Hall International, 1989

A.M.Turing: “Checking a Large Routine”, Cambridge University, Report on a Conference on High Speed Automatic Calculating Machines, pages 67,..70, 1949

End of Bibliography

11.3 Index

End of Index

11.4 Laws

11.4.0 Booleans

Let a, b, c, d, and e be boolean.

Boolean Laws

$\top$

$\neg \bot$

Law of Excluded Middle (Tertium non Datur)

$a \vee \neg a$

Law of Noncontradiction

$\neg(a \wedge \neg a)$

Base Laws

$\neg(a \wedge \bot)$

$a \vee \top$

$a \Rightarrow \top$

$\bot \Rightarrow a$

Identity Laws

$\top \wedge a = a$

$\bot \vee a = a$

$\top \Rightarrow a = a$

$\top = a = a$

Idempotent Laws

$a \wedge a = a$

$a \vee a = a$

Reflexive Laws

$a \Rightarrow a$

$a = a$

Laws of Indirect Proof

$\neg a \Rightarrow \bot = a$ (Reductio ad Absurdum)

$\neg a \Rightarrow a = a$

Law of Specialization

$a \wedge b \Rightarrow a$

Associative Laws

$a \wedge (b \wedge c) = (a \wedge b) \wedge c$

$a \vee (b \vee c) = (a \vee b) \vee c$

$a = (b = c) = (a = b) = c$

$a \neq (b \neq c) = (a \neq b) \neq c$

$a = (b \neq c) = (a = b) \neq c$

Law of Double Negation

$\neg \neg a = a$

Duality Laws (deMorgan)

$\neg(a \wedge b) = \neg a \vee \neg b$

$\neg(a \vee b) = \neg a \wedge \neg b$

Laws of Exclusion

$a \Rightarrow \neg b = b \Rightarrow \neg a$

$a = \neg b = a \neq b = \neg a = b$

Laws of Inclusion

$a \Rightarrow b = \neg a \vee b$ (Material Implication)

$a \Rightarrow b = (a \wedge b = a)$

$a \Rightarrow b = (a \vee b = b)$

Absorption Laws

$a \wedge (a \vee b) = a$

$a \vee (a \wedge b) = a$

Laws of Direct Proof

$(a \Rightarrow b) \wedge a \Rightarrow b$ (Modus Ponens)

$(a \Rightarrow b) \wedge \neg b \Rightarrow \neg a$ (Modus Tollens)

$(a \vee b) \wedge \neg a \Rightarrow b$(Disjunctive Syllogism)

Transitive Laws

$(a \wedge b) \wedge (b \wedge c) \Rightarrow (a \wedge c)$

$(a \Rightarrow b) \wedge (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$

$(a = b) \wedge (b = c) \Rightarrow (a = c)$

$(a \Rightarrow b) \wedge (b = c) \Rightarrow (a \Rightarrow c)$

$(a = b) \wedge (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$

Distributive Laws (Factoring)

$a \wedge (b \wedge c) = (a \wedge b) \wedge (a \wedge c)$

$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

$a \vee (b \vee c) = (a \vee b) \vee (a \vee c)$

$a \vee (b \Rightarrow c) = (a \vee b) \Rightarrow (a \vee c)$

$a \vee (b = c) = (a \vee b) = (a \vee c)$

$a \Rightarrow (b \wedge c) = (a \Rightarrow b) \wedge (a \Rightarrow c)$

$a \Rightarrow (b \vee c) = (a \Rightarrow b) \vee (a \Rightarrow c)$

$a \Rightarrow (b \Rightarrow c) = (a \Rightarrow b) \Rightarrow (a \Rightarrow c)$

$a \Rightarrow (b = c) = (a \Rightarrow b) = (a \Rightarrow c)$

Symmetry Laws (Commutative Laws)

$a \wedge b = b \wedge a$
$a \vee b = b \vee a$
$a = b = b = a$
$a \neq b = b \neq a$

Antisymmetry Law (Double Implication)

$(a \Rightarrow b) \wedge (b \Rightarrow a) = a = b$

Laws of Discharge

$a \wedge (a \Rightarrow b) = a \wedge b$
$a \Rightarrow (a \wedge b) = a \Rightarrow b$

Antimonotonic Law

$a \Rightarrow b \Longrightarrow (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$

Contrapositive Law

$a \Rightarrow b = \neg b \Rightarrow \neg a$

Law of Generalization

$a \Rightarrow a \vee b$

Antidistributive Laws

$a \wedge b \Rightarrow c = (a \Rightarrow c) \vee (b \Rightarrow c)$
$a \vee b \Rightarrow c = (a \Rightarrow c) \wedge (b \Rightarrow c)$

Laws of Portation

$a \wedge b \Rightarrow c = a \Rightarrow (b \Rightarrow c)$
$a \wedge b \Rightarrow c = a \Rightarrow \neg b \vee c$

Laws of Conflation

$(a \Rightarrow b) \wedge (c \Rightarrow d) \Longrightarrow a \wedge c \Rightarrow b \wedge d$
$(a \Rightarrow b) \wedge (c \Rightarrow d) \Longrightarrow a \vee c \Rightarrow b \vee d$

Monotonic Laws

$a \Rightarrow b \Longrightarrow c \wedge a \Rightarrow c \wedge b$
$a \Rightarrow b \Longrightarrow c \vee a \Rightarrow c \vee b$
$a \Rightarrow b \Longrightarrow (c \Rightarrow a) \Rightarrow (c \Rightarrow b)$

Law of Resolution

$a \wedge c \Longrightarrow (a \vee b) \wedge (\neg b \vee c) = (a \wedge \neg b) \vee (b \wedge c) \Longrightarrow a \vee c$

Case Base Laws

if $\top$ **then** a **else** $b = a$
if $\bot$ **then** a **else** $b = b$

One Case Laws

if a **then** b **else** $\top = a \Rightarrow b$
if a **then** b **else** $\bot = a \wedge b$

Case Reversal Law

if a **then** b **else** c
$=$ **if** $\neg a$ **then** c **else** b

Case Absorption Laws

if a **then** b **else** $c =$ **if** a **then** $a \wedge b$ **else** c
if a **then** b **else** $c =$ **if** a **then** $a \Rightarrow b$ **else** c
if a **then** b **else** $c =$ **if** a **then** $a = b$ **else** c
if a **then** b **else** $c =$ **if** a **then** b **else** $\neg a \wedge c$
if a **then** b **else** $c =$ **if** a **then** b **else** $a \vee c$
if a **then** b **else** $c =$ **if** a **then** b **else** $a \neq c$

Case Distributive Laws (Case Factoring)

$\neg$ **if** a **then** b **else** $c =$ **if** a **then** $\neg b$ **else** $\neg c$
(**if** a **then** b **else** c) $\wedge d =$ **if** a **then** $b \wedge d$ **else** $c \wedge d$
and similarly replacing $\wedge$ by any of $\vee = \neq \Rightarrow \Leftarrow$
if a **then** $b \wedge c$ **else** $d \wedge e =$ (**if** a **then** b **else** d) $\wedge$ (**if** a **then** c **else** e)
and similarly replacing $\wedge$ by any of $\vee = \neq \Rightarrow \Leftarrow$

End of Booleans

11.4.1 Generic

The operators $= \neq$ **if then else** apply to every type of expression, with the laws

$x = x$	reflexivity
$x=y \;=\; y=x$	symmetry
$x=y \wedge y=z \;\Rightarrow\; x=z$	transitivity
$x=y \;\Rightarrow\; f\,x = f\,y$	transparency
$x\neq y \;=\; \neg(x=y)$	unequality
if $\top$ **then** x **else** $y \;=\; x$	case base
if $\bot$ **then** x **else** $y \;=\; y$	case base

The operators $< \leq > \geq$ apply to numbers, characters, strings, and lists, with the laws

$\neg\, x<x$	irreflexivity
$\neg(x<y \wedge x>y)$	exclusivity
$\neg(x<y \wedge x=y)$	exclusivity
$x\leq y \wedge y\leq x = x=y$	antisymmetry
$x<y \wedge y<z \Rightarrow x<z$	transitivity
$x\leq y \;=\; x<y \vee x=y$	inclusivity
$x>y \;=\; y<x$	mirror
$x\geq y \;=\; y\leq x$	mirror
$x<y \vee x=y \vee x>y$	totality, trichotomy

End of Generic

11.4.2 Numbers

Let d be a sequence of (zero or more) digits, and let x, y, and z be numbers.

$d0+1 = d1$	counting
$d1+1 = d2$	counting
$d2+1 = d3$	counting
$d3+1 = d4$	counting
$d4+1 = d5$	counting
$d5+1 = d6$	counting
$d6+1 = d7$	counting
$d7+1 = d8$	counting
$d8+1 = d9$	counting
$d9+1 = (d+1)0$	counting (see Exercise 22)
$x+0 = x$	identity
$x+y = y+x$	symmetry
$x+(y+z) \;=\; (x+y)+z$	associativity
$-\infty<x<\infty \Rightarrow (x+y = x+z \;=\; y=z)$	cancellation
$-\infty<x \;\Rightarrow\; \infty+x = \infty$	absorption
$x<\infty \;\Rightarrow\; -\infty + x = -\infty$	absorption
$-x \;=\; 0 - x$	negation
$- -x = x$	self-inverse
$-(x+y) = -x + -y$	distributivity
$-(x-y) = -x - -y$	distributivity
$-(x\times y) = -x \times y$	semi-distributivity
$-(x/y) = -x \,/\, y$	semi-distributivity
$x-y = -(y-x)$	antisymmetry
$x-y \;=\; x + -y$	subtraction

$x + (y - z) = (x + y) - z$	associativity
$-\infty<x<\infty \Rightarrow (x-y = x-z = y=z)$	cancellation
$-\infty<x<\infty \Rightarrow x-x = 0$	inverse
$x<\infty \Rightarrow \infty-x = \infty$	absorption
$-\infty<x \Rightarrow -\infty - x = -\infty$	absorption
$-\infty<x<\infty \Rightarrow x\times0 = 0$	base
$x\times1 = x$	identity
$x\times y = y\times x$	symmetry
$x\times(y+z) = x\times y + x\times z$	distributivity
$x\times(y\times z) = (x\times y)\times z$	associativity
$-\infty<x<\infty \wedge x\neq0 \Rightarrow (x\times y = x\times z = y=z)$	cancellation
$0<x \Rightarrow x\times\infty = \infty$	absorption
$0<x \Rightarrow x \times -\infty = -\infty$	absorption
$x/1 = x$	identity
$-\infty<x<\infty \wedge x\neq0 \Rightarrow x/x = 1$	inverse
$x\times(y/z) = (x\times y)/z = x/(z/y)$	multiplication-division
$y\neq0 \Rightarrow (x/y)/z = x/(y\times z)$	multiplication-division
$-\infty<x<\infty \Rightarrow x/\infty = 0 = x/-\infty$	annihilation
$-\infty<x<\infty \Rightarrow x^0 = 1$	base
$x^1 = x$	identity
$x^{y+z} = x^y \times x^z$	exponents
$x^{y\times z} = (x^y)^z$	exponents
$-\infty<0<1<\infty$	direction
$x<y = -y<-x$	reflection
$-\infty<x<\infty \Rightarrow (x+y < x+z = y<z)$	cancellation, translation
$0<x<\infty \Rightarrow (x\times y < x\times z = y<z)$	cancellation, scale
$x<y \vee x=y \vee x>y$	trichotomy
$-\infty \leq x \leq \infty$	extremes

End of Numbers

11.4.3 Bunches

Let x and y be elements (booleans, numbers, characters, sets, strings and lists of elements).

$x: y = x=y$	elementary law
$x: A,B = x: A \vee x: B$	compound law
$A,A = A$	idempotence
$A,B = B,A$	symmetry
$A,(B,C) = (A,B),C$	associativity
$A‘A = A$	idempotence
$A‘B = B‘A$	symmetry
$A‘(B‘C) = (A‘B)‘C$	associativity
$A,B: C = A: C \wedge B: C$	antidistributivity
$A: B‘C = A: B \wedge A: C$	distributivity
$A: A,B$	generalization
$A‘B: A$	specialization
$A: A$	reflexivity
$A: B \wedge B: A = A=B$	antisymmetry
$A: B \wedge B: C \Rightarrow A: C$	transitivity
$¢\ null = 0$	size
$¢x = 1$	size

$¢(A, B) + ¢(A`B) = ¢A + ¢B$	size
$¬\ x{:}\ A \Rightarrow ¢(A`x) = 0$	size
$A{:}\ B \Rightarrow ¢A \le ¢B$	size
$A,(A`B) = A$	absorption
$A`(A,B) = A$	absorption
$A{:}\ B = A,B = B = A = A`B$	inclusion
$A,(B,C) = (A,B),(A,C)$	distributivity
$A,(B`C) = (A,B)`(A,C)$	distributivity
$A`(B,C) = (A`B), (A`C)$	distributivity
$A`(B`C) = (A`B)`(A`C)$	distributivity
$A{:}\ B \wedge C{:}\ D \Rightarrow A,C{:}\ B,D$	conflation, monotonicity
$A{:}\ B \wedge C{:}\ D \Rightarrow A`C{:}\ B`D$	conflation, monotonicity
$null{:}\ A$	induction
$A, null = A$	identity
$A\ `\ null = null$	base
$¢A = 0 = A = null$	size
$x{:}\ int \wedge y{:}\ xint \wedge x \le y \Rightarrow (i{:}\ x,..y = i{:}\ int \wedge x \le i < y)$	
$x{:}\ int \wedge y{:}\ xint \wedge x \le y \Rightarrow ¢(x,..y) = y–x$	
$–null = null$	distribution
$–(A, B) = –A, –B$	distribution
$A+null = null+A = null$	distribution
$(A, B)+(C, D) = A+C, A+D, B+C, B+D$	distribution

and similarly for many other operators (see the final page of the book)

End of Bunches

11.4.4 Sets

	$\{A\}{:}\ ↯B = A{:}\ B$
$\{\sim S\} = S$	$\$\{A\} = ¢A$
$\sim\{A\} = A$	$\{A\} \cup \{B\} = \{A, B\}$
$\{A\} \ne A$	$\{A\} \cap \{B\} = \{A\ `\ B\}$
$A \in \{B\} = A{:}\ B$	$\{A\} = \{B\} = A = B$
$\{A\} \subseteq \{B\} = A{:}\ B$	$\{A\} \ne \{B\} = A \ne B$

End of Sets

11.4.5 Strings

Let S, T, and U be strings; let i and j be items (booleans, numbers, characters, bunch of items, sets, lists, functions); let n be extended natural; let x, y, and z be integers.

$nil; S = S; nil = S$	$↔S<\infty \Rightarrow nil \le S < S; i; T$
$S; (T; U) = (S; T); U$	$↔S<\infty \Rightarrow (i<j = S; i; T < S; j; U)$
$↔nil = 0$	$↔S<\infty \Rightarrow (i=j = S; i; T = S; j; T)$
$↔i = 1$	$0*S = nil$
$↔(S; T) = ↔S + ↔T$	$(n+1)*S = n*S; S$
$S_{nil} = nil$	$↔S<\infty \Rightarrow S;i;T \triangleleft ↔S \triangleright j = S;j;T$
$↔S<\infty \Rightarrow (S; i; T)_{↔S} = i$	$x;..x = nil$
$S_{T; U} = S_T; S_U$	$x;..x+1 = x$
$S_{(T_U)} = (S_T)_U$	$(x;..y)\ ;\ (y;..z) = x;..z$
$S_{\{A\}} = \{S_A\}$	$↔(x;..y) = y–x$

End of Strings

11.4.6 Lists

Let S and T be strings; let i and j be items (booleans, numbers, characters, bunch of items, sets, lists, functions); let L, M, and N be lists.

$[S] \neq S$
$\sim[S] = S$
$[\sim L] = L$
$[S]\ T = S_T$
$[S]+[T] = [S; T]$
$[S] = [T] \;=\; S = T$
$[S] < [T] \;=\; S < T$
$nil \rightarrow i \mid L = i$
$n \rightarrow i \mid [S] = [S \lhd n \rhd i]$
$(S;T) \rightarrow i \mid L = S \rightarrow (T \rightarrow i \mid L@S) \mid L$

$\#[S] = \leftrightarrow S$
$S_{[T]} = [S_T]$
$[S]\ [T] = [S_T]$
$L\ \{A\} = \{L\ A\}$
$L\ [S] = [L\ S]$
$(L\ M)\ N = L\ (M\ N)$
$L@nil = L$
$L@i = L\ i$
$L@(S; T) = L@S@T$

End of Lists

11.4.7 Functions

Renaming Law — if v and w do not appear in D and w does not appear in b
$\langle v{:}\ D \rightarrow b \rangle = \langle w{:}\ D \rightarrow \langle v{:}\ D \rightarrow b \rangle w \rangle$

Application Law: if element $x{:}\ D$
$\langle v{:}\ D \rightarrow b \rangle x =$ (substitute x for v in b)

Law of Extension
$f = \langle w{:}\ \Delta f \rightarrow f\ w \rangle$

Domain Law
$\Delta \langle v{:}\ D \rightarrow b \rangle = D$

Function Composition Laws: If $\neg\ f{:}\ \Delta g$
$\Delta(g\ f) = \S x{:}\ \Delta f{\cdot}\ fx{:}\ \Delta g$
$(g\ f)\ x = g\ (f\ x)$
$f\ (g\ h) = (f\ g)\ h$

Function Inclusion Law
$f{:}\ g \;=\; \Delta g{:}\ \Delta f \wedge \forall x{:}\ \Delta g{\cdot}\ fx{:}\ gx$

Cardinality Law
$\not\subset A = \Sigma\ (A \rightarrow 1)$

Function Equality Law
$f = g \;=\; \Delta f = \Delta g \wedge \forall x{:}\ \Delta f{\cdot}\ fx = gx$

Laws of Functional Intersection
$\Delta(f\ `\ g) = \Delta f, \Delta g$
$(f\ `\ g)\ x = (f \mid g)\ x\ `\ (g \mid f)\ x$

Laws of Functional Union
$\Delta(f, g) = \Delta f\ `\ \Delta g$
$(f, g)\ x = f\ x, g\ x$

Laws of Selective Union
$\Delta(f \mid g) = \Delta f, \Delta g$
$(f \mid g)\ x =$ **if** $x{:}\ \Delta f$ **then** $f\ x$ **else** $g\ x$
$f \mid (g \mid h) = (f \mid g) \mid h$

Laws of Selective Union
$f \mid f = f$
$(g \mid h)\ f = g\ f \mid h\ f$
$\langle v{:}\ A \rightarrow x \rangle \mid \langle v{:}\ B \rightarrow y \rangle = \langle v{:}\ A, B \rightarrow$ **if** $v{:}\ A$ **then** x **else** $y \rangle$

Distributive Laws
$f\ null = null$
$f\ (A, B) = f\ A, f\ B$
$f\ (\S g) = \S y{:}\ f\ (\Delta g){\cdot}\ \exists x{:}\ \Delta g{\cdot}\ fx{=}y \wedge gx$
$f\ (\textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y) = \textbf{if}\ b\ \textbf{then}\ f\ x\ \textbf{else}\ f\ y$
$(\textbf{if}\ b\ \textbf{then}\ f\ \textbf{else}\ g)\ x = \textbf{if}\ b\ \textbf{then}\ f\ x\ \textbf{else}\ g\ x$

Arrow Laws
$f{:}\ null \rightarrow A$
$A \rightarrow B{:}\ (A\ `C) \rightarrow (B,D)$
$f{:}\ A \rightarrow B \;=\; A{:}\ \Delta f \wedge \forall a{:}\ A{\cdot}\ fa{:}\ B$

End of Functions

11.4.8 Quantifiers

Let x be an element, let a, b and c be boolean, let n and m be numeric, let f and g be functions, and let P be a predicate.

$\forall v\colon null\cdot b = \top$
$\forall v\colon x\cdot b = \langle v\colon x\to b\rangle\, x$

$\forall v\colon A,B\cdot b = (\forall v\colon A\cdot b) \wedge (\forall v\colon B\cdot b)$
$\forall v\colon (\S v\colon D\cdot b)\cdot c = \forall v\colon D\cdot b \Rightarrow c$

$\exists v\colon null\cdot b = \bot$
$\exists v\colon x\cdot b = \langle v\colon x\to b\rangle\, x$

$\exists v\colon A,B\cdot b = (\exists v\colon A\cdot b) \vee (\exists v\colon B\cdot b)$
$\exists v\colon (\S v\colon D\cdot b)\cdot c = \exists v\colon D\cdot b \wedge c$

$\Sigma v\colon null\cdot n = 0$
$\Sigma v\colon x\cdot n = \langle v\colon x\to n\rangle\, x$

$(\Sigma v\colon A,B\cdot n) + (\Sigma v\colon A\text{`}B\cdot n) = (\Sigma v\colon A\cdot n) + (\Sigma v\colon B\cdot n)$
$\Sigma v\colon (\S v\colon D\cdot b)\cdot n = \Sigma v\colon D\cdot$ **if** b **then** n **else** 0

$\Pi v\colon null\cdot n = 1$
$\Pi v\colon x\cdot n = \langle v\colon x\to n\rangle\, x$

$(\Pi v\colon A,B\cdot n) \times (\Pi v\colon A\text{`}B\cdot n) = (\Pi v\colon A\cdot n) \times (\Pi v\colon B\cdot n)$
$\Pi v\colon (\S v\colon D\cdot b)\cdot n = \Pi v\colon D\cdot$ **if** b **then** n **else** 1

$MIN\ v\colon null\cdot n = \infty$
$MIN\ v\colon x\cdot n = \langle v\colon x\to n\rangle\, x$

$MIN\ v\colon A,B\cdot n = min\ (MIN\ v\colon A\cdot n)\ (MIN\ v\colon B\cdot n)$
$MIN\ v\colon (\S v\colon D\cdot b)\cdot n = MIN\ v\colon D\cdot$ **if** b **then** n **else** ∞

$MAX\ v\colon null\cdot n = -\infty$
$MAX\ v\colon x\cdot n = \langle v\colon x\to n\rangle\, x$

$MAX\ v\colon A,B\cdot n = max\ (MAX\ v\colon A\cdot n)\ (MAX\ v\colon B\cdot n)$
$MAX\ v\colon (\S v\colon D\cdot b)\cdot n = MAX\ v\colon D\cdot$ **if** b **then** n **else** $-\infty$

$\S v\colon null\cdot b = null$
$\S v\colon x\cdot b =$ **if** $\langle v\colon x\to b\rangle\, x$ **then** x **else** $null$
$\S v\colon A,B\cdot b = (\S v\colon A\cdot b), (\S v\colon B\cdot b)$
$\S v\colon A\text{`}B\cdot b = (\S v\colon A\cdot b)\ \text{`}\ (\S v\colon B\cdot b)$
$\S v\colon (\S v\colon D\cdot b)\cdot c = \S v\colon D\cdot b \wedge c$

Change of Variable Laws — if d does not appear in b

$\forall r\colon fD\cdot b = \forall d\colon D\cdot \langle r\colon fD\to b\rangle\ (fd)$
$\exists r\colon fD\cdot b = \exists d\colon D\cdot \langle r\colon fD\to b\rangle\ (fd)$
$\Sigma r\colon fD\cdot n = \Sigma d\colon D\cdot \langle r\colon fD\to n\rangle\ (fd)$
$\Pi r\colon fD\cdot n = \Pi d\colon D\cdot \langle r\colon fD\to n\rangle\ (fd)$
$MIN\ r\colon fD\cdot n = MIN\ d\colon D\cdot \langle r\colon fD\to n\rangle\ (fd)$
$MAX\ r\colon fD\cdot n = MAX\ d\colon D\cdot \langle r\colon fD\to n\rangle\ (fd)$

Identity Laws

$\forall v\cdot \top$
$\neg\exists v\cdot \bot$

Bunch-Element Conversion Laws

$V\colon W = \forall v\colon V\cdot \exists w\colon W\cdot v{=}w$
$fV\colon gW = \forall v\colon V\cdot \exists w\colon W\cdot fv{=}gw$

Idempotent Laws — if $D{\neq}null$ and v does not appear in b

$\forall v\colon D\cdot b = b$
$\exists v\colon D\cdot b = b$

Absorption Laws — if $x\colon D$

$\langle v\colon D\to b\rangle\, x \wedge \exists v\colon D\cdot b = \langle v\colon D\to b\rangle\, x$
$\langle v\colon D\to b\rangle\, x \vee \forall v\colon D\cdot b = \langle v\colon D\to b\rangle\, x$
$\langle v\colon D\to b\rangle\, x \wedge \forall v\colon D\cdot b = \forall v\colon D\cdot b$
$\langle v\colon D\to b\rangle\, x \vee \exists v\colon D\cdot b = \exists v\colon D\cdot b$

Distributive Laws — if $D{\neq}null$ and v does not appear in a

$a \wedge \forall v\colon D\cdot b = \forall v\colon D\cdot a \wedge b$
$a \wedge \exists v\colon D\cdot b = \exists v\colon D\cdot a \wedge b$
$a \vee \forall v\colon D\cdot b = \forall v\colon D\cdot a \vee b$
$a \vee \exists v\colon D\cdot b = \exists v\colon D\cdot a \vee b$
$a \Rightarrow \forall v\colon D\cdot b = \forall v\colon D\cdot a \Rightarrow b$
$a \Rightarrow \exists v\colon D\cdot b = \exists v\colon D\cdot a \Rightarrow b$

Antidistributive Laws — if $D{\neq}null$ and v does not appear in a

$a \Leftarrow \exists v\colon D\cdot b = \forall v\colon D\cdot a \Leftarrow b$
$a \Leftarrow \forall v\colon D\cdot b = \exists v\colon D\cdot a \Leftarrow b$

Specialization Law — if x: D

$$\forall v\colon D\cdot b \implies \langle v\colon D\to b\rangle x$$

Generalization Law — if x: D

$$\langle v\colon D\to b\rangle\, x \implies \exists v\colon D\cdot b$$

One-Point Laws — if element x: D and v does not appear in x

$$\forall v\colon D\cdot v{=}x \Rightarrow b \;=\; \langle v\colon D\to b\rangle\, x$$
$$\exists v\colon D\cdot v{=}x \wedge b \;=\; \langle v\colon D\to b\rangle\, x$$

Duality Laws

$$\neg\forall v\cdot b \;=\; \exists v\cdot \neg b \quad \text{(deMorgan)}$$
$$\neg\exists v\cdot b \;=\; \forall v\cdot \neg b \quad \text{(deMorgan)}$$
$$-\mathit{MAX}\ v\cdot n \;=\; \mathit{MIN}\ v\cdot -n$$
$$-\mathit{MIN}\ v\cdot n \;=\; \mathit{MAX}\ v\cdot -n$$

Solution Laws — if x is an element

$$\S v\colon D\cdot \top \;=\; D$$
$$(\S v\colon D\cdot b)\colon\ D$$
$$\S v\colon D\cdot \bot \;=\; \mathit{null}$$
$$(\S v\cdot b)\colon (\S v\cdot c) \;=\; \forall v\cdot b{\Rightarrow}c$$
$$(\S v\cdot b), (\S v\cdot c) \;=\; \S v\cdot b \vee c$$
$$(\S v\cdot b)\ \text{`}\ (\S v\cdot c) \;=\; \S v\cdot b \wedge c$$
$$x\colon \S p \;=\; x\colon \Delta p \;\wedge\; px$$
$$\forall f \;=\; (\S f){=}(\Delta f)$$
$$\exists f \;=\; (\S f){\neq}\mathit{null}$$

Bounding Laws — if v does not appear in n

$$n > (\mathit{MAX}\ v\colon D\cdot m) \;\Rightarrow\; (\forall v\colon D\cdot n{>}m)$$
$$n < (\mathit{MIN}\ v\colon D\cdot m) \;\Rightarrow\; (\forall v\colon D\cdot n{<}m)$$
$$n \geq (\mathit{MAX}\ v\colon D\cdot m) \;=\; (\forall v\colon D\cdot n{\geq}m)$$
$$n \leq (\mathit{MIN}\ v\colon D\cdot m) \;=\; (\forall v\colon D\cdot n{\leq}m)$$
$$n \geq (\mathit{MIN}\ v\colon D\cdot m) \;\Leftarrow\; (\exists v\colon D\cdot n{\geq}m)$$
$$n \leq (\mathit{MAX}\ v\colon D\cdot m) \;\Leftarrow\; (\exists v\colon D\cdot n{\leq}m)$$
$$n > (\mathit{MIN}\ v\colon D\cdot m) \;=\; (\exists v\colon D\cdot n{>}m)$$
$$n < (\mathit{MAX}\ v\colon D\cdot m) \;=\; (\exists v\colon D\cdot n{<}m)$$

Splitting Laws — for any fixed domain

$$\forall v\cdot a \wedge b \;=\; (\forall v\cdot a) \wedge (\forall v\cdot b)$$
$$\exists v\cdot a \wedge b \;\Rightarrow\; (\exists v\cdot a) \wedge (\exists v\cdot b)$$
$$\forall v\cdot a \vee b \;\Leftarrow\; (\forall v\cdot a) \vee (\forall v\cdot b)$$
$$\exists v\cdot a \vee b \;=\; (\exists v\cdot a) \vee (\exists v\cdot b)$$
$$\forall v\cdot a \Rightarrow b \;\Rightarrow\; (\forall v\cdot a) \Rightarrow (\forall v\cdot b)$$
$$\forall v\cdot a \Rightarrow b \;\Rightarrow\; (\exists v\cdot a) \Rightarrow (\exists v\cdot b)$$
$$\forall v\cdot a = b \;\Rightarrow\; (\forall v\cdot a) = (\forall v\cdot b)$$
$$\forall v\cdot a = b \;\Rightarrow\; (\exists v\cdot a) = (\exists v\cdot b)$$

Commutative Laws

$$\forall v\cdot \forall w\cdot b \;=\; \forall w\cdot \forall v\cdot b$$
$$\exists v\cdot \exists w\cdot b \;=\; \exists w\cdot \exists v\cdot b$$

Semicommutative Laws (Skolem)

$$\exists v\cdot \forall w\cdot b \;\Rightarrow\; \forall w\cdot \exists v\cdot b$$
$$\forall x\cdot \exists y\cdot Pxy \;=\; \exists f\cdot \forall x\cdot Px(fx)$$

Domain Change Laws

$$A\colon B \;\Rightarrow\; (\forall v\colon A\cdot b) \;\Leftarrow\; (\forall v\colon B\cdot b)$$
$$A\colon B \;\Rightarrow\; (\exists v\colon A\cdot b) \;\Rightarrow\; (\exists v\colon B\cdot b)$$
$$\forall v\colon A\cdot v\colon B \Rightarrow p \;=\; \forall v\colon A\text{`}B\cdot p$$
$$\exists v\colon A\cdot v\colon B \wedge p \;=\; \exists v\colon A\text{`}B\cdot p$$

Extreme Law

$$\forall v\cdot (\mathit{MIN}\ v\cdot n) \leq n \leq (\mathit{MAX}\ v\cdot n)$$

Connection Laws (Galois)

$$n{\leq}m \;=\; \forall k\cdot\ k{\leq}n \Rightarrow k{\leq}m$$
$$n{\leq}m \;=\; \forall k\cdot\ k{<}n \Rightarrow k{<}m$$
$$n{\leq}m \;=\; \forall k\cdot\ m{\leq}k \Rightarrow n{\leq}k$$
$$n{\leq}m \;=\; \forall k\cdot\ m{<}k \Rightarrow n{<}k$$

Distributive Laws — if $D{\neq}\mathit{null}$ and v does not appear in n

$$\mathit{max}\ n\ (\mathit{MAX}\ v\colon D\cdot m) \;=\; (\mathit{MAX}\ v\colon D\cdot \mathit{max}\ n\ m)$$
$$\mathit{max}\ n\ (\mathit{MIN}\ v\colon D\cdot m) \;=\; (\mathit{MIN}\ v\colon D\cdot \mathit{max}\ n\ m)$$
$$\mathit{min}\ n\ (\mathit{MAX}\ v\colon D\cdot m) \;=\; (\mathit{MAX}\ v\colon D\cdot \mathit{min}\ n\ m)$$
$$\mathit{min}\ n\ (\mathit{MIN}\ v\colon D\cdot m) \;=\; (\mathit{MIN}\ v\colon D\cdot \mathit{min}\ n\ m)$$
$$n + (\mathit{MAX}\ v\colon D\cdot m) \;=\; (\mathit{MAX}\ v\colon D\cdot n{+}m)$$
$$n + (\mathit{MIN}\ v\colon D\cdot m) \;=\; (\mathit{MIN}\ v\colon D\cdot n{+}m)$$
$$n - (\mathit{MAX}\ v\colon D\cdot m) \;=\; (\mathit{MIN}\ v\colon D\cdot n{-}m)$$
$$n - (\mathit{MIN}\ v\colon D\cdot m) \;=\; (\mathit{MAX}\ v\colon D\cdot n{-}m)$$
$$(\mathit{MAX}\ v\colon D\cdot m) - n \;=\; (\mathit{MAX}\ v\colon D\cdot m{-}n)$$
$$(\mathit{MIN}\ v\colon D\cdot m) - n \;=\; (\mathit{MIN}\ v\colon D\cdot m{-}n)$$
$$n{\geq}0 \;\Rightarrow\; n \times (\mathit{MAX}\ v\colon D\cdot m) \;=\; (\mathit{MAX}\ v\colon D\cdot n{\times}m)$$
$$n{\geq}0 \;\Rightarrow\; n \times (\mathit{MIN}\ v\colon D\cdot m) \;=\; (\mathit{MIN}\ v\colon D\cdot n{\times}m)$$
$$n{\leq}0 \;\Rightarrow\; n \times (\mathit{MAX}\ v\colon D\cdot m) \;=\; (\mathit{MIN}\ v\colon D\cdot n{\times}m)$$
$$n{\leq}0 \;\Rightarrow\; n \times (\mathit{MIN}\ v\colon D\cdot m) \;=\; (\mathit{MAX}\ v\colon D\cdot n{\times}m)$$
$$n \times (\Sigma v\colon D\cdot m) \;=\; (\Sigma v\colon D\cdot n{\times}m)$$
$$(\Pi v\colon D\cdot m)^n \;=\; (\Pi v\colon D\cdot m^n)$$

End of Quantifiers

11.4.9 Limits

$(MAX\ m\cdot\ MIN\ n\cdot\ f(m+n))\ \le\ (LIM\ f)\ \le\ (MIN\ m\cdot\ MAX\ n\cdot\ f(m+n))$
$\exists m\cdot\ \forall n\cdot\ p(m+n)\ \Rightarrow\ LIM\ p\ \Rightarrow\ \forall m\cdot\ \exists n\cdot\ p(m+n)$

End of Limits

11.4.10 Specifications and Programs

For specifications P, Q, R, and S, and boolean b,

$ok\ =\ x'=x\ \wedge\ y'=y\ \wedge\ ...$
$x:=e\ =\ x'=e\ \wedge\ y'=y\ \wedge\ ...$
$P.\ Q\ =\ \exists x'', y'', ...\cdot\ \langle x', y', ...\rightarrow P\rangle\ x''\ y''\ ...\ \wedge\ \langle x, y, ...\rightarrow Q\rangle\ x''\ y''\ ...$
$P\|Q\ =\ \exists t_P, t_Q\cdot\ \langle t'\rightarrow P\rangle t_P\ \wedge\ \langle t'\rightarrow Q\rangle t_Q\ \wedge\ t'=max\ t_P\ t_Q$
if b **then** P **else** $Q\ =\ b\wedge P\vee\neg b\wedge Q$
var $x: T\cdot\ P\ =\ \exists x, x': T\cdot\ P$
frame $x\cdot\ P\ =\ P\ \wedge\ y'=y\ \wedge\ ...$
while b **do** $P\ =\ t'\ge t\wedge$ (**if** b **then** $(P.\ t:=t+1.$ **while** b **do** $P)$ **else** $ok)$
$\forall\sigma, \sigma'\cdot$ (**if** b **then** $(P.\ W)$ **else** $ok\ \Leftarrow\ W)\ \Rightarrow\ \forall\sigma, \sigma'\cdot$ (**while** b **do** $P\ \Leftarrow\ W)$
$\quad (Fmn\ \Leftarrow\ m=n\wedge ok)\wedge(Fik\ \Leftarrow\ m\le i<j<k\le n\wedge(Fij.\ Fjk))$
$\Rightarrow\quad (Fmn\ \Leftarrow$ **for** $i:=m;..n$ **do** $m\le i<n\Rightarrow Fi(i+1))$
$Im\Rightarrow I'n\ \Leftarrow$ **for** $i:=m;..n$ **do** $m\le i<n\wedge Ii\Rightarrow I'(i+1)$
wait until $w\ =\ t:=max\ t\ w$
assert $b\ =$ **if** b **then** ok **else** ($print$ "error". **wait until** ∞)
ensure $b\ =\ b\wedge ok$
$x'=(P$ **result** $e)\ =\ P.\ x'=e$
$c?\ =\ r:=r+1$
$c\ =\ \mathbb{M}c_{\,rc-1}$
$c!\ e\ =\ \mathbb{M}c_{\,wc}=e\ \wedge\ \mathbb{T}c_{\,wc}=t\ \wedge\ (wc:=wc+1)$
$\surd c\ =\ \mathbb{T}c_{\,rc}+$ (transit time) $\le t$
ivar $x: T\cdot\ S\ =\ \exists x: time\rightarrow T\cdot\ S$
chan $c: T\cdot\ P\ =\ \exists\mathbb{M}c: \infty*T\cdot\ \exists\mathbb{T}c: \infty*xreal\cdot$ **var** $rc, wc: xnat := 0\cdot\ P$

$ok.\ P\ =\ P.\ ok\ =\ P$	identity
$P.\ (Q.\ R)\ =\ (P.\ Q).\ R$	associativity
if b **then** P **else** $P\ =\ P$	idempotence
if b **then** P **else** $Q\ =$ **if** $\neg b$ **then** Q **else** P	case reversal
$P\ =$ **if** b **then** $b\Rightarrow P$ **else** $\neg b\Rightarrow P$	case creation
$P\vee Q.\ R\vee S\ =\ (P.\ R)\vee(P.\ S)\vee(Q.\ R)\vee(Q.\ S)$	distributivity
(**if** b **then** P **else** $Q).\ R\ =$ **if** b **then** $(P.\ R)$ **else** $(Q.\ R)$	distributivity (unprimed b)
$ok\ \|\|\ P\ =\ P\ \|\|\ ok\ =\ P$	identity
$P\ \|\|\ Q\ =\ Q\ \|\|\ P$	symmetry
$P\ \|\|\ (Q\ \|\|\ R)\ =\ (P\ \|\|\ Q)\ \|\|\ R$	associativity
$P\ \|\|\ Q\vee R\ =\ (P\ \|\|\ Q)\vee(P\ \|\|\ R)$	distributivity
$P\ \|\|$ **if** b **then** Q **else** $R\ =$ **if** b **then** $(P\ \|\|\ Q)$ **else** $(P\ \|\|\ R)$	distributivity
if b **then** $(P\|\|Q)$ **else** $(R\|\|S)\ =$ **if** b **then** P **else** $R\ \|\|$ **if** b **then** Q **else** S	distributivity
$x:=$ **if** b **then** e **else** $f\ =$ **if** b **then** $x:=e$ **else** $x:=f$	functional-imperative

End of Specifications and Programs

11.4.11 Substitution

Let x and y be different boundary state variables, let e and f be expressions of the prestate, and let P be a specification.

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

$(x{:=}\,e \parallel y{:=}\,f).\ P$ = (for x substitute e and independently for y substitute f in P)

End of Substitution

11.4.12 Conditions

Let P and Q be any specifications, and let C be a precondition, and let C' be the corresponding postcondition (in other words, C' is the same as C but with primes on all the state variables).

$C \wedge (P.\,Q) \Leftarrow C \wedge P.\,Q$

$C \Rightarrow (P.Q) \Leftarrow C \Rightarrow P.\,Q$

$(P.Q) \wedge C' \Leftarrow P.\,Q \wedge C'$

$(P.Q) \Leftarrow C' \Leftarrow P.\,Q \Leftarrow C'$

$P.\,C \wedge Q \Leftarrow P \wedge C'.\,Q$

$P.\,Q \Leftarrow P \wedge C'.\ C \Rightarrow Q$

C is a sufficient precondition for P to be refined by S
if and only if $C \Rightarrow P$ is refined by S.

C' is a sufficient postcondition for P to be refined by S
if and only if $C' \Rightarrow P$ is refined by S.

End of Conditions

11.4.13 Refinement

Refinement by Steps (Stepwise Refinement) (monotonicity, transitivity)

If $A \Leftarrow$ **if** b **then** C **else** D and $C \Leftarrow E$ and $D \Leftarrow F$ are theorems,
then $A \Leftarrow$ **if** b **then** E **else** F is a theorem.

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

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

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

Refinement by Parts (monotonicity, conflation)

If $A \Leftarrow$ **if** b **then** C **else** D and $E \Leftarrow$ **if** b **then** F **else** G are theorems,
then $A \wedge E \Leftarrow$ **if** b **then** $C \wedge F$ **else** $D \wedge G$ is a theorem.

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.

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

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

Refinement by Cases

$P \Leftarrow$ **if** b **then** Q **else** R is a theorem if and only if
$P \Leftarrow b \wedge Q$ and $P \Leftarrow \neg b \wedge R$ are theorems.

End of Refinement

End of Laws

11.5 Names

abs: *xreal*→§*r*: *xreal*· *r*≥0	*abs r* = **if** *r*≥0 **then** *r* **else** –*r*
bool (the booleans)	*bool* = ⊤, ⊥
ceil: *real*→*int*	*r* ≤ *ceil r* < *r*+1
char (the characters)	*char* = ..., "a", "A", ...
div: *real*→(§*r*: *real*· *r*>0)→*int*	*div x y* = *floor* (*x*/*y*)
divides: (*nat*+1)→*int*→*bool*	*divides n i* = *i*/*n*: *int*
entro: *prob*→§*r*: *xreal*· *r*≥0	*entro p* = *p* × *info p* + (1–*p*) × *info* (1–*p*)
even: *int*→*bool*	*even i* = *i*/2: *int*
	even = *divides* 2
floor: *real*→*int*	*floor r* ≤ *r* < *floor r* + 1
info: *prob*→§*r*: *xreal*· *r*≥0	*info p* = – *log p*
int (the integers)	*int* = *nat*, –*nat*
LIM (limit quantifier)	see Laws
log: (§*r*: *xreal*· *r*≥0)→*xreal*	*log* (2^*x*) = *x*
	log (*x*×*y*) = *log x* + *log y*
max: *xrat*→*xrat*→*xrat*	*max x y* = **if** *x*≥*y* **then** *x* **else** *y*
	– *max a b* = *min* (–*a*) (–*b*)
MAX (maximum quantifier)	see Laws
min: *xrat*→*xrat*→*xrat*	*min x y* = **if** *x*≤*y* **then** *x* **else** *y*
	– *min a b* = *max* (–*a*) (–*b*)
MIN (minimum quantifier)	see Laws
mod: *real*→(§*r*: *real*· *r*>0)→*real*	0 ≤ *mod a d* < *d*
	a = *div a d* × *d* + *mod a d*
nat (the naturals)	0, *nat*+1: *nat*
	0, *B*+1: *B* ⇒ *nat*: *B*
nil (the empty string)	↔*nil* = 0
	nil; *S* = *S* = *S*; *nil*
	nil ≤ *S*
null (the empty bunch)	¢*null* = 0
	null, *A* = *A* = *A*, *null*
	null: *A*
odd: *int*→*bool*	*odd i* = ¬ *i*/2: *int*
	odd = ¬*even*
ok (the empty program)	*ok* = σ′=σ
	ok.*P* = *P*.*ok* = *ok* ‖ *P* = *P* ‖ *ok* = *P*
prob (probability)	*prob* = §*r*: *real*· 0≤*r*≤1
rand (random number)	*rand n*: 0,..*n*
rat (the rationals)	*rat* = *int*/(*nat*+1)
real (the reals)	*r*: *real* = *r*: *xreal* ∧ –∞<*r*<∞
suc: *nat*→(*nat*+1)	*suc n* = *n*+1
xint (the extended integers)	*xint* = –∞, *int*, ∞
xnat (the extended naturals)	*xnat* = *nat*, ∞
xrat (the extended rationals)	*xrat* = –∞, *rat*, ∞
xreal (the extended reals)	*x*: *xreal* = ∃*f*: *nat*→*rat*· *x* = *LIM f*

End of Names

11.6 Symbols

⊤	3	true
⊥	3	false
¬	3	not
∧	3	and
∨	3	or
⇒	3	implies
⟹	3	implies
⇐	3	follows from, is implied by
⟸	3	follows from, is implied by
=	3	equals, if and only if
⩵	3	equals, if and only if
≠	3	differs from, is unequal to
<	13	less than
>	13	greater than
≤	13	less than or equal to
≥	13	greater than or equal to
+	12	plus
⁺	20	list catenation
–	12	minus
×	12	times, multiplication
/	12	divided by
,	14	bunch union
,..	16	union from (incl) to (excl)
‘	14	bunch intersection
;	17	string catenation
;..	19	catenation from (incl) to (excl)
:	14	is in, are in, bunch inclusion
::	89	includes
:=	36	assignment
.	36	dep't (sequential) composition
·	26	quantifier abbreviation
!	133	output
?	133	input
√	133	input check

( )	4	parentheses for grouping
{ }	17	set brackets
[ ]	20	list brackets
⟨ ⟩	23	function (scope) brackets
↯	17	power
¢	14	bunch size, cardinality
$	17	set size, cardinality
↔	18	string size, length
#	20	list size, length
|	20,24	selective union, otherwise
∥	118	indep't (parallel) composition
~	17	contents of a set
⍨	20	contents of a list
*	18	repetition of a string
Δ	23	domain of a function
→	23	function arrow
∈	17	element of a set
⊆	17	subset
∪	17	set union
∩	17	set intersection
@	22	index with a pointer
∀	26	for all, universal quantifier
∃	26	there exists, existential quantifier
Σ	26	sum of, summation quantifier
Π	26	product of, product quantifier
§	28	those, solution quantifier
′	34	x' is final value of state var x
"	13,19	"hello" is a text or string of chars
a^b	12	exponentiation
a_b	18	string indexing
a b	20,31	indexing,application,composition
◁ ▷	18	string modification
∞	12	infinity

assert	77	**ivar**	126
chan	138	**loop end**	71
ensure	77	**or**	77
exit when	71	**result**	78
for do	74	**var**	66
frame	67	**wait until**	76
go to	76	**while do**	69
if then else	4		

End of Symbols

11.7 Precedence

0 ⊤ ⊥ ( ) { } [ ] ⟨ ⟩ **loop end** numbers characters texts names
1 @ juxtaposition
2 prefix– ¢ $ ↔ # * ~ ¬ ↯ Δ → √ superscript subscript
3 × / ∩
4 + infix– ⁺ ∪
5 ; ;.. ‘
6 , ,... | ◁ ▷
7 = ≠ < > ≤ ≥ : :: ∈ ⊆
8 ¬
9 ∧
10 ∨
11 ⇒ ⇐
12 := ! ?
13 **if then else while do exit when for do go to wait until assert ensure or**
14 . ‖ **result**
15 ∀· ∃· Σ· Π· §· *LIM*· *MAX*· *MIN*· **var**· **ivar**· **chan**· **frame**·
16 ⩵ ⟹ ⟸

On level 2, superscripting and subscripting serve to bracket all operations within them.

Juxtaposition associates from left to right, so $a\ b\ c$ means the same as $(a\ b)\ c$. The infix operators @ / – associate from left to right. The infix operators * → associate from right to left. The infix operators × ∩ + ⁺ ∪ ; ‘ , | ∧ ∨ . ‖ are associative (they associate in both directions).

On levels 7, 11, and 16 the operators are continuing. For example, $a = b = c$ neither associates to the left nor associates to the right, but means the same as $a = b \wedge b = c$. On any one of these levels, a mixture of continuing operators can be used. For example, $a \le b < c$ means the same as $a \le b \wedge b < c$.

On levels 13 and 15, the precedence applies to the final operand (and to both operands of **or**), not to operands that are surrounded by the operator.

The operators ⩵ ⟹ ⟸ are identical to = ⇒ ⇐ except for precedence.

End of Precedence

11.8 Distribution

The operators in the following expressions distribute over bunch union in any operand:

$[A]$ $A@B$ $A\ B$ $-A$ $\$A$ $\leftrightarrow A$ $\#A$ $\sim A$ $\neg A$
A^B A_B $A \times B$ A/B $A \cap B$ $A+B$ $A-B$ A^+B $A \cup B$ $A;B$ $A‘B$
$\neg A$ $A \wedge B$ $A \vee B$

The operator in A^*B distributes over bunch union in its left operand only.

End of Distribution

End of Reference

End of a Practical Theory of Programming