

While loops and programs

In these notes we discuss one of the most common programming constructs, the while loop. To do this we will set up a simple programming language *While* in which the loop is the only non-trivial construct. We will then analyse various aspects of this language. The while loop is a very powerful tool. It is analogous to the μ -operator, unbounded search, in recursion theory. Thus, although the language *While* is syntactically simple, it is still very strong. (A real life programming language in this style will have more features, but these merely make it more convenient to use rather than increase its strength.) *While* is an imperative language, that is it uses **assignments** and has an associated notion of **state**. (These concepts will be explained later.) There are other styles of programming, and one of these – the **functional** style – is dealt with using λ - calculi.

Contents

1	Introduction	1
	Exercises	4
2	The machine \mathfrak{M} and language <i>While</i>	5
	Exercises	14
3	Denotational semantics	15
	Exercises	23
4	Head and tail recursion	25
	Exercises	28
5	More refined recursions	30
	Exercises	35
	Some solutions	36

1 Introduction

This topic of these notes is part of a much wider subject, and it is worth considering some of the generalities before with look at the particular aspects.

Suppose we have a specification of a function

$$\mathbb{S} \xrightarrow{f} \mathbb{T}$$

for some source type \mathbb{S} and target type \mathbb{T} . Given an input $s \in \mathbb{S}$ how can we evaluate f at s , that is determine the output $fs \in \mathbb{T}$? More precisely, how can we write a program to do this job?

There are many kinds of specifications, but here we will restrict our attention to those which are recursive in some sense. In other words, by inspecting

the specification it will be possible to extract an informal algorithm for evaluating the function. Our problem is to format that algorithm as a *While*-program.

Even for this restricted case, there are several points to think about.

In general a specification need not have a unique solution. However, for the kinds we consider here, a recursive specification has a unique **least solution**. This solution may be partial, that is a total function of the form

$$\mathbb{D} \xrightarrow{f_D} \mathbb{T}$$

where $\mathbb{D} \subseteq \mathbb{S}$ is the domain of definition of the function. Furthermore, each other solution will extend f_D , that is will be a total function of the form

$$\mathbb{E} \xrightarrow{f_E} \mathbb{T}$$

where $\mathbb{D} \subseteq \mathbb{E} \subseteq \mathbb{S}$ with $f_E s = f_D s$ for $s \in \mathbb{D}$. It is the function f_D we wish to program.

A second point is that the algorithm embedded in a specification may not be very efficient. For example, think of the various ways of specifying the binomial coefficients. An explicit definition

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

could lead to a lot of waste if the three values $n!, i!, (n-i)!$ are calculated separately and then combined. An algorithm based on Pascal's triangle is more efficient, provided the temporary values are stored in a reasonable way.

(Usually, in a first course of recursive functions, this particular function is shown to be recursive by coding certain lists of values as one number. Often the properties of prime factorization are used for this. However, this trick in no good if we want to program the function in any reasonable way. We need to kind of device for storing lists of values. The easiest way to handle this is to develop an arithmetic lists with various recursive constructs and program dicetect with names for such lists. Because of lack of time we can't we don't explain the details of this here.)

The job of a recursive specification is to isolate one particular partial function, it need not provide an efficient algorithm. Part of the job of converting a recursive specification into a program is to turn one algorithm into a more efficient one.

Another point to be thought about is how the members of \mathbb{S} and \mathbb{T} should be notated. For each run of the program certain values have to be stored (in the machine on which the program is run). When the values are natural numbers, integers, or even rational numbers, the appropriate notations are fairly straight forward. However, if the values are real numbers, then there are several possible notations that could be used, and some are better than others for certain jobs.

We won't worry too much about this aspect. However, you should remember that both \mathbb{S} and \mathbb{T} may be tuples. That is, f may have the form

$$\mathbb{S}_1 \times \cdots \times \mathbb{S}_s \xrightarrow{f} \mathbb{T}_1 \times \cdots \times \mathbb{T}_t$$

for certain components $\mathbb{S}_1, \dots, \mathbb{T}_t$. This means that f is a batch of functions

$$\mathbb{S}_1 \times \cdots \times \mathbb{S}_s \xrightarrow{f_j} \mathbb{T}_j \quad 1 \leq j \leq t$$

which must be evaluated in parallel. Thus the corresponding program may have to manipulate many different kinds of values.

The final, and perhaps most important, point is that of **verification**. Once the program is written there should be a proof that it does the job it is supposed to do. That is, there should be a proof that (when used in the correct way) it does compute a function, and this function is the canonical solution of the parent specification. The program should come with a certificate of correctness.

(Most software firms, being shyster organizations, don't provide certificates. If a program they have provided goes wrong then they will charge you again for correcting their mistake. In fact, providing a certificate for anything beyond the simplest program is very difficult. The subject is in its infancy; analogous to the state of calculus before the Newton-Leibnitz era.)

In these notes we consider how a recursive specification, of the kind we have met in **Forms of recursion**, can be converted into a program, and how that program can be verified. Naturally, since recursion is just refined iteration (do this until that isn't the case) the verification technique will use induction. Most of the problem is setting up the induction in an appropriate way. The programming construct we use, the while loop, is just a variant of the refined iteration (while that is the case do this). Having just read this there may be something puzzling you here. There is a subtlety over the meaning of 'iteration' which will be explained when we look at while loops in detail.

Each programming language is designed in a certain way to run on a particular kind of machine which has certain features and operates in certain way. This is an idealized machine, not an actual machine. In the next section we first give a brief, informal description of such a machine, and then we give a description of the associated programming language *While*.

Each program is a piece of syntax which must satisfy certain 'rules of grammar'. The process of associating a meaning to syntax is **semantics**. In section 3 we consider this process for *While*. This is the **denotational semantics** of the language. (There are various other activities with names like 'operational semantics', 'axiomatic semantics', 'silly bugger semantics' which, in the strict sense, are not semantics at all, but analyse other aspects of the behaviour of the language. We will not deal with those here, but we will look at an operational behaviour in λ -calculi.)

Sections 2 and 3 deal mostly with generalities. In sections 4 and 5 we look at some particular forms of recursive specifications.¹

To conclude this preamble there is an important point to be explained. This is rarely, if ever, addressed in accounts of this material.

From the work of Turing and others in the 1930s we may identify the informally computable functions with those computable on a Turing (or similar) machine. Furthermore, this is exactly the class of general recursive functions. Part of the proof of this equivalence shows how to convert a description of a function as general recursive into a program for evaluating the function on the machine. That seems to be what we want to do here. The recursive specification will identify the function as general recursive, so there will be a method for translating that into a program. Perhaps the particular relevant details have never been written down, but we know they are out there somewhere.

It is the case that any general recursive function is programmable in *While*, and so this particular language is just as or more powerful as any other programming language. This is true, but misses the point.

There are many different characterizations of the Turing computable functions. For any two styles of characterization there is a proof that the two are equivalent. Each of these results can be stated as follows.

Each function which is the unique solution of a specification in style A is also the unique solution of a specification in style B.

The proof will show how to translate any given A-specification into some B-specification with the same solution. Unfortunately, it is *not* the functions we are concerned with, but the algorithms for evaluating the functions. Each translation, from A to B, will use a lot of coding which will completely destroy all of the important facets of the source algorithm (such as efficiency and cost). What we want to do here is produce more delicate translations of an algorithm into an algorithm which does not destroy the important facets, and even improves those facets.

Exercises

1 This exercise show how to generate Pascal's triangle by manipulating lists. Here a list

$[a, b, \dots]$

is a finite sequence of natural numbers. It is convenient to let a, b, \dots range over natural numbers, and let l, m, \dots range over list. We write $[]$ for the empty list and al for the list form from l by pushing a onto the leading position. This increases the length of the list by 1. Let \mathbb{L} be the set of all these lists.

¹In fact, section 5 is unfinished and was not be dealt with in the lectures. This material was repalced by an introduction to domain theory. However, for convenience the first draft of that section has been left in these notes.

Consider the transition relation

$$\mathbb{N} \times \mathbb{N} \times \mathbb{L} \times \mathbb{L} \longrightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{L} \times \mathbb{L}$$

given by

$$\begin{aligned} (0) \quad & (r, 0, l, []) \longmapsto (0, 0, [], []) \\ (1) \quad & (r, 0, l, bm) \longmapsto (r', b, m, [1]) \\ (2) \quad & (r, b, [], m) \longmapsto (r, 0, [], bm) \\ (3) \quad & (r, b, al, m) \longmapsto (r, a, l, (b+a)m) \end{aligned}$$

for $r, a, b \in \mathbb{N}$ and $l, m \in \mathbb{N}$. because of clause (0) this transitions can be repeated indefinitely. We may think of a 4-tuple

$$(r, b, l, m)$$

as a state, so the transition converts each state into a state.

(a) Convince yourself that starting from the state

$$(0, 0, [], [1])$$

the transition path will generate all the information in Pascal's triangle.

(b) Using this transition, write an informal program to calculate $\binom{n}{i}$.

2 The machine \mathfrak{M} and language *While*

In this section we first describe an abstract machine which can be used to manipulate collections of values. We then describe a programming language *While* which can be used to guide the behaviour of the machine. That is, we show how to write a simple program which can be run on the machine.

The machine \mathfrak{M} has two parts.

- A potentially infinite collection of **registers** each of which can hold a value or be empty.
- A processing unit which can combine values in certain simple ways.

You will often find such a device described as an **idealized** machine. This mean two things. Firstly, there is no limit an the amount of storage in the machine. Here there is no limit on the number of registers we may use, and each can hold as much data as we want. Thus, if the job of a particular register is to hold a natural number, then that number can be as big as we like, bigger than the universe if necessary. Secondly, there is no limit on how long the machine may run before it gives an answer, even if that answer is only 42.

Clearly, we could design and build an actual machine along the lines suggested, but we would have to put a limit on the number of stores and the length of a run.

We use upper case Roman letters such as X, Y, Z, \dots to denote registers. Each program will be a finite piece of syntax which will make reference to certain of these registers. Thus each program can use just finitely many registers. However, the machine is capable of providing as many registers as are needed.

Each register can hold a value or be empty. In any program the values held by a particular register must be of a certain type as required by the the program. If a register X is used in a program, then there will be a restriction

$$X : \mathbb{X}$$

where \mathbb{X} is some concrete type of values. This means that X may hold only (names of) members of \mathbb{X} or be empty. At any stage in any run of a program, if the register X is non-empty, then it must hold a member of \mathbb{X} . Thus, in any one program each register (that is used) is assigned a type.

For instance, in a program we may want to use the natural numbers \mathbb{N} , the integers \mathbb{Z} , the rational numbers \mathbb{Q} , or any other space we want to calculate with. Almost always we will want to use the rather trivial space \mathbb{B} of boolean values (which has just two members). We should not get these various concrete types confused.

It is useful to begin each program with a **program header**

$$\text{PROG}[X_1 : \mathbb{X}_1, \dots, X_s : \mathbb{X}_s]$$

giving its name **PROG** and listing all the registers used at some point in the program together with the type of values that can be held by each register.

For instance, if a program is designed to evaluate a function

$$\mathbb{N}, \mathbb{P} \xrightarrow{f} \mathbb{T}$$

then the header will contain the requirements

$$X : \mathbb{N}, P : \mathbb{P}, T : \mathbb{T}$$

indicating that

- the register X can only hold a natural number (and in the first instance will be used to hold the input x when evaluating $f(x, p)$)
- the register P can only hold a parameter (and in the first instance will be used to hold the input p when evaluating $f(x, p)$)
- the register T can only hold a member of \mathbb{T} (and in the final instance will be used to hold the eventual output $f(x, p)$)

are the restrictions on (and part of the job of) the registers. Of course, in the middle of a run of the program the registers may hold values different from the ones indicated, but each must agree with its given type. Also the program may use other registers to hold various other values that might be needed in the calculation (such as the number of steps taken until now, or some previously calculated value that might be needed later.)

Consider a program **PROG** using certain types $\mathbb{X}_1, \dots, \mathbb{X}_s$, as above. At the start of a run of **PROG** each register $X : \mathbb{X}$ will hold a value $x \in \mathbb{X}$ or be empty. Similarly, at the end of a run each register $X : \mathbb{X}$ will hold a value $x^* \in \mathbb{X}$ or be empty. This is the idea behind the **denotational semantics** of **PROG**. We think of the program as (a name of a gadget) which transforms s -tuples into s -tuples (where there are type restrictions on these s -tuples). A program is like a recursive specification for a transition function, and so encodes an algorithm for calculating the transitions. (This may seem a bit circular, but more explanation will be given later.)

Because of the possibility of empty registers, there is a snag.

At any point in a run of **PROG**, each register $X : \mathbb{X}$ must hold a value $x \in \mathbb{X}$ or be empty. Thus the transition function is partial, and we have to find some way of handling this. We use a simple trick. We adjoin to \mathbb{X} a tag \perp which is intended to be a nominal value ‘empty’. Thus we work with

$$\mathbb{X}_\perp = \mathbb{X} \cup \{\perp\}$$

rather than \mathbb{X} . This is an idea from **domain theory**, and the construction results in a **lifted domain** \mathbb{X}_\perp . Of course, we don’t use a tag that is already a member of \mathbb{X} .

Using this trick we can convert a partial function into a total function. In particular, we can view the meaning of a program as a certain **state transformer**. We go into the details of this in section 3, but the informal idea can be given here.

Consider a program **PROG** as above. This uses certain types $\mathbb{X}_1, \dots, \mathbb{X}_s$. A **state** for **PROG** is an s -tuple

$$(x_1, \dots, x_s)$$

where $x_i \in \mathbb{X}_i \cup \{\perp\}$ for each $1 \leq i \leq s$. Let **State** be the set of such states. The **meaning** of **PROG** is a certain function

$$\text{State} \longrightarrow \text{State}$$

generated from the syntax of the program.

You may have noticed earlier, and perhaps by now forgotten, that so far we have said nothing about the syntax of a program beyond its header. We can begin that now.

An atomic action of the machine is to take the values $x_1 \in \mathbb{X}_1, \dots, x_r \in \mathbb{X}_r$ currently stored in certain registers, combine them in a certain way to produce a value $h(x_1, \dots, x_r)$ and store this in some nominated register Y of the correct

type. This process destroys any value that might be already held in Y , but the contents of the other registers are preserved.

This atomic step is indicated by

$$Y := h(X_1, \dots, X_r)$$

and is an **assignment**. This should not be read as an equality. For example, an assignment

$$X := X + 1$$

increments the contents of X by 1, but the equality $X = X + 1$ is meaningless.

Where does this function h come from in an assignment? The processing unit has some simple facilities built into it. These functions are usually of the order of addition or multiplication, nothing very sophisticated. Otherwise we may invoke a previously written program designed to evaluate h . In short, we may call a subroutine.

Assignments are put together in blocks and separated by ‘;’ to indicate they are to be executed in sequence. Sometimes these have to be read with a little care.

2.1 EXAMPLE. At first sight it seems that the block

$$\begin{aligned} X &:= Y; \\ Y &:= Z; \\ Z &:= X \end{aligned}$$

produces a cyclic permutation

$$(x, y, z) \longmapsto (y, z, x)$$

of the three values held. In fact, the atomic steps are

$$(x, y, z) \longmapsto (y, y, z) \longmapsto (y, z, z) \longmapsto (y, z, y)$$

so that the value x is lost. To achieve a permutation a temporary register is needed. ■

As in this example blocks are often written vertically. This is because the syntax of a longish program needs to be visually structured, and vertical alignments are a useful punctuation device. A block can be written horizontally

$$X := Y; Y := Z; Z := X$$

if it is not too long and its internal workings are not too important.

The intended meaning of an assignment and a sequential composition are obvious, but we will give a formal definition later.

It is clear that a block of assignments can not achieve much. Essentially, all it can calculate is various composites of the data functions called. Thus,

in order to carry out more sophisticated calculations we need more powerful constructs.

Here we use a **while loop**. This is an instruction to repeatedly perform a given instruction until a certain condition is met, and then stop (or go onto the next phase of the program). The syntax we use is

$$\mathit{while} B \mathit{do} Q \mathit{od}$$

where

- B is an expression which take boolean values (that is it is either true or false)
- Q is an instruction (which itself may include some while loops)

and the ‘ od ’ is a punctuation device to show the end of this particular loop. While loops may be nested and we need some kind of device to indicate where one loop ends and another begins.

A few simple examples will help.

2.2 EXAMPLES. Consider the following three programs where

$$[X : \mathbb{N}, Y : \mathbb{Z}, Z : \mathbb{Z}, I : \mathbb{N}]$$

is the header for each.

$\frac{\text{FAC}[- -]}{Z := Y; I := 0; \mathit{while} I < X \mathit{do} \quad I := I + 1; \quad Z := Z \times I \quad \mathit{od}}$	$\frac{\text{EXP}[- -]}{Z := 1; I := X; \mathit{while} I \neq 0 \mathit{do} \quad Z := Y \times Z; \quad I := I - 1 \quad \mathit{od}}$	$\frac{\text{MLT}[- -]}{I := 0; \mathit{while} X \neq 0 \mathit{do} \quad Z := Y + Z; \quad X := X - 1 \quad \mathit{od}}$
--	---	--

By running each of these we find that the behaviour is

$$(x, y, \cdot, \cdot) \xrightarrow{\text{FAC}} (x, y, x! \times y, x)$$

$$(x, y, \cdot, \cdot) \xrightarrow{\text{EXP}} (x, y, y^x, 0)$$

$$(x, y, z, \cdot) \xrightarrow{\text{MLT}} (x, y, xy + z, 0)$$

respectively. All three use X, Y as input registers and Z as an output register. The third also uses Z as an input. All three use I as an internal, counting register. ■

You should run these programs to see what happens, and verify the claims in the example. Notice that they do not make the most efficient use of registers. Also, you should try changing the order of the two assignments in the body of the while loop, and observe the effect. (When I say ‘run the programs’ you can do this with pencil and paper. If you have never seen this set up before it is instructive to draw little boxes for the registers and track what happens to them as each instruction is obeyed. Alternatively, you can build a machine out of match boxes and knicker elastic and send it off to Blue Peter.)

This more or less gives a full description of *While*, except for one extra construct which we include to make the examples a bit more interesting.

In the remainder of this section we show what a formal definition of the language looks like (without labouring the details). After that, in the following sections, we will begin the real work of organizing the semantics of the language, and consider how we might verify that a program is correct.

It is useful to divide *While* into three syntactic categories, that is three classes of syntactic constructions where each class is designed to do a particular kind of job. These are as follows.

- There is an unlimited stock of **identifiers**. These are just characters (or strings of characters) and each is used to name (that is identify) a register. They are sometimes called program variables (because the people who first chose the name only had a limited grasp of the idea of a variable).
- There is a collection of **expressions** built up from identifiers in several predetermined ways. Each expression is a template for combining certain identifiers to produce a value relative to the values taken by the identifiers. It will not be necessary to set down just how expressions are built.
- There is a collection of **instructions** built in a way to be described. Each instruction has a meaning which is a **state transformer**.

It is useful to have some informal conventions for these three syntactic categories. Thus we let

$$\begin{aligned} X, Y, Z \dots & \text{ range over identifiers} \\ A, B, C \dots & \text{ range over expressions} \\ P, Q, R \dots & \text{ range over instructions} \end{aligned}$$

where at times it is convenient to break these conventions.

The job of an identifier is to name a register which may hold values. However, as we have seen, for any particular program there is a restriction on the type of these values. We formalize this idea. (The same idea is used to a much greater degree in the typed λ -calculus.)

2.3 DEFINITION. A **declaration** is a pair $X : \mathbb{X}$ where X is an identifier and \mathbb{X} is a type (that is a space of values).

A context is a list

$$X_1 : \mathbb{X}_1, \dots, X_s : \mathbb{X}_s$$

of declarations

$$X : \mathbb{X}$$

where the identifiers X_1, \dots, X_s are pairwise distinct. ■

We use upper case Greek letters $\Gamma, \Delta, \Pi, \dots$ to range over contexts.

Notice that a subtle change has been made here. Previously we used pairs $X : \mathbb{X}$ where X is a register and \mathbb{X} is a type. Now such a pair has an identifier in place of the register. All we have done is replace the abstract entity, the register, by a piece of syntax, the identifier, which is the name of the register.

The expressions of the language are a bit like generalized polynomials. They are built up from the identifiers using the names of various operations built into the language or, in our case, the names of functions that are calculated elsewhere.

2.4 EXAMPLE. Suppose the language can use a certain binary operation \odot . This will live out in the real world but will have a name in the language, say \square . With these names for operations the **expressions** will be generated something like the following.

- Each identifier is an expression.
- ...
- If B and C are expressions (of the correct type) then so is $(B \square C)$.
- ...

There is a base clause (picking out the identifiers), and then a step clause for each acceptable way of combining smaller expressions to form larger expressions. ■

This informal idea will be enough for what we do here. However, it is not the whole story, and it is worth looking briefly at a vital missing component.

Suppose, in the example above, the external operation is

$$\mathbb{B}, \mathbb{C} \xrightarrow{\odot} \mathbb{T}$$

where the two inputs types \mathbb{B}, \mathbb{C} might not be the same. Thus the operation produces a value $b \odot c$ *only* if $b \in \mathbb{B}, c \in \mathbb{C}$. If b, c come from elsewhere then the combination $b \odot c$ is meaningless. (For instance, when working with a vector space we have two kinds of values, scalars and vectors. These can be combined in various ways. We can add two scalars and we can add two vectors, but we can not add a scalar to a vector. The same symbol ‘+’ is used for two different operation. This is sometimes described as ‘overloading the syntax’.)

In Example 2.4, the construction of $(B \boxdot C)$ appears to allow B, C to be any expressions. In fact, because it is not too important for what we do here, we have missed out part of the construction. There ought to be an associated **typing discipline**. The phrase in brackets ‘(of the correct type)’ is an indication that something extra should be done. This is dealt with in detail when we look at λ -calculi, but here is a partial description of could be done for *While*.

Remember that the construction of an expression is suppose to take place within a context

$$\Gamma = [X_1 : \mathbb{X}_1, \dots, X_s : \mathbb{X}_s]$$

which assigns to each identifier a range of variation, its type. We allow these restrictions to permeate through the construction of expressions. Thus, rather than generate a raw expression, as in the example, we generate a judgement

$$\Gamma \vdash A : \mathbb{A}$$

which can be read as ‘within the context Γ the raw expression A is correctly formed to name a value in \mathbb{A} ’. More correctly, we named an operation

$$\mathbb{X}_1 \times \dots \times \mathbb{X}_s \longrightarrow \mathbb{A}$$

which is built up in a certain way out of the primitive operations (such as \odot).

We can rephrase the construction rules for raw expressions to produce judgements which take account of the obvious typing restrictions.

The base clause is

$$\Gamma \vdash X : \mathbb{X}$$

where $X : \mathbb{X}$ is a declaration in Γ . The step clause for \boxdot is

$$\frac{\Gamma \vdash B : \mathbb{B} \quad \Gamma \vdash C : \mathbb{C}}{\Gamma \vdash (B \boxdot C) : \mathbb{T}}$$

which say the provided B and C are correctly formed, then so is $(B \boxdot C)$.

For what we do here the typing restrictions are ‘obvious’ so we may safely assume that they have been dealt with and get on with a more important part of the analysis. (If you are reading this a second time you will see that a typing discipline can be produced along the line of a simply typed λ -calculus without the need for abstractions. The generated judgements can be given a categorical semantics but, because of the presence of \perp , a category of domains is needed.)

The construction of expressions is nothing more than the syntactic counterpart of the construction of composite functions from certain primitive functions. The important part of the programming language is the way it formats the orders that certain actions are to be carried out.

2.5 DEFINITION. The instructions of the language are built in four ways.

- If X is an identifier and A is an expression, then the **assignment**

$$X := A$$

is an instruction.

- If Q and R are instructions then the **sequential composite**

$$Q; R$$

is an instruction.

- If B is an expression and Q is an instruction, then the **while loop**

$$\textit{while } B \textit{ do } Q \textit{ od}$$

is an instruction.

- If B is an expression and Q and R are instructions, then the **conditional**

$$\textit{if } B \textit{ then do } Q \textit{ else do } R \textit{ fi}$$

is an instruction.

There are no other instructions. ■

This definition shows how to generate each instruction by recursion over the syntax. There are atomic instructions, the assignments, and then each instruction is built from these using a sequence of three other constructs. You should note that this generates the syntactically correct instructions, but only a subclass of these can have a meaning.

Clearly, for a loop or a conditional to be meaningful, the expression B must produce boolean values. Something like

$$\textit{while } 17 \textit{ do } Q \textit{ od}$$

is meaningless. We can get rid of these ‘non-instructions’ by extending the typing discipline mentioned above. That makes the language more complicated, and doesn’t help much for what we do here, so we won’t bother.

2.6 DEFINITION. A **program** is just an instruction in context, that is a pair $\Gamma|P$ where Γ is a context and P is an instruction and where each identifier used in P is declared in Γ . ■

For a program $\Gamma|P$ in this sense the important part is the instruction P . The context Γ is little more than a convenient listing of the identifiers. However, if we imposed a typing discipline on *While* then for $\Gamma|P$ to be a program the instruction would have to be well formed in the context Γ .

At this stage it will help if you look at some simple programs with two kinds of questions in mind. Firstly, given a specification of a function (or batch of functions), how can that be turned into a program? Secondly, given a program, how can we work out what it is doing? Neither of these questions are as easy as they might seem.

Exercises

2 In Examples 2.2 we saw a program which can be used to compute the exponential function $(x, y) \mapsto y^x$ for $x \in \mathbb{N}, y \in \mathbb{Z}$. Here are three more programs which do a similar job.

$\frac{[X, Y, Z : \mathbb{N}]}{Z := 1;}$ <pre style="margin: 0;"> while X ≠ 0 do Z := Z × Y; X := X - 1 od </pre>	$\frac{[X, Y, Z : \mathbb{N}]}{Z := 1;}$ <pre style="margin: 0;"> while X ≠ 0 do if X is even then do X := X/2; Y := Y² else do Z := Z × Y; X := X - 1 fi od </pre>	$\frac{[X, Y, Z : \mathbb{N}]}{Z := 1;}$ <pre style="margin: 0;"> while X ≠ 0 do while X is even do X := X/2; Y := Y² od; Z := Z × Y; X := X - 1 od </pre>
---	--	---

In each case describe how the program works, and write down the transition function.

Can you see what is unchanged on each pass through the loop?

What can you say about the efficiency of these programs?

3 Consider the following (non-recursive) specification of a pair of functions

$$\mathbb{N}, \mathbb{N} \xrightarrow{\text{quo}} \mathbb{N} \quad \mathbb{N}, \mathbb{N} \xrightarrow{\text{rem}} \mathbb{N}$$

the quotient and remainder functions. These are sometimes called ‘div’ and ‘mod’.

For all $x, y \in \mathbb{N}$ if $y \neq 0$ then

$$x = qy + r$$

where $0 \leq r < y$ and

$$q = \text{quo}(x, y) \quad r = \text{rem}(x, y)$$

are the two values used.

Notice there are no restrictions on the functions for $y = 0$.

Using the obvious algorithm (that is, by testing each of $0, y, 2y, 3y, \dots$ in turn) write a program with a single while loop to calculate the two functions.

Can you make this program more efficient?

4 The generalized fibonacci function

$$\mathbb{Z}, \mathbb{R}, \mathbb{R}, \mathbb{R}, \mathbb{R} \xrightarrow{f} \mathbb{R}$$

is specified by

$$\begin{aligned} f(0, a, b, x, y) &= x \\ f(1, a, b, x, y) &= y \\ f(r + 2, a, b, x, y) &= af(r + 1, a, b, x, y) + bf(r, a, b, x, y) \end{aligned}$$

for $r \in \mathbb{Z}, a, b, x, y \in \mathbb{R}$.

- (a) Show that this specification defines a unique function, and this is total.
- (b) Write a program to evaluate this function.

You might like to first consider the function restricted to $r \in \mathbb{N}$.

5 Write a program which, when supplied with a positive real $x \in \mathbb{R}$ will calculate $\lfloor \sqrt{x} \rfloor$, the integer part of the square root of x .

3 Denotational semantics

In section 2 we set up the syntax of the programming language and gave an informal description of what its constructs are supposed to mean. In this section we will look at the details of this meaning, and to do that we must work more formally.

Each program $\Gamma|P$ is a context Γ and an instruction P . The job of the context Γ is to give the identifiers used in the instruction P and say what type of value each identifier X can take. Thus Γ is a list of declarations $X : \mathbb{X}$ indicating that the identifier X may be used somewhere in P but can only take values from \mathbb{X} (or be empty). Remember that we allow \perp to be a nominal value (to indicate ‘empty’). We say \mathbb{X} is the **housing type** of the identifier X .

A state for Γ assigns to each identifier $X : \mathbb{X}$ a value taken from the domain $\mathbb{X}_\perp = \mathbb{X} \cup \{\perp\}$. Previously we bundled this up as a tuple, but because of what we are going to do it is more convenient to think of this as a function (of a certain kind).

3.1 DEFINITION. Let Γ be a context (for a program). Let $\mathcal{I}d = \mathcal{I}d(\Gamma)$ be the set of identifiers declared in Γ , and for each $X \in \mathcal{I}d$ let $\mathbb{X}(X)$ be the housing type of X . Thus we have a declaration $X : \mathbb{X}(X)$ in Γ , and

$$(\mathbb{X}(X) \mid X \in \mathcal{I}d)$$

is the indexed family of types used in Γ . There may be some repetitions in this family. Let

$$\mathbb{T} = \coprod (\mathbb{X}(X)_\perp \mid X \in \mathcal{I}d)$$

be the disjoint union of the indexed family of lifted domains (with repetitions counted).

A **state** over Γ is a function

$$\sigma : \mathcal{I}d \longrightarrow \mathbb{T}$$

where

$$\sigma X \in \mathbb{X}(X)_\perp$$

for each $X \in \mathcal{I}d$.

Let **State** = **State**(Γ) be the space of states over Γ . ■

Thus a state is a function σ which assigns to each identifier X a value σX which lives in the appropriate place as declared in the context. Here a ‘value’ may be \perp , which indicates that the register named by corresponding identifier is empty. If you think about it, a state in this sense is nothing more than a tuple as before. We set up a state in this way because of the manipulation we will go through. In particular, we can compose a state σ with other functions. However, in examples we will continue to write out a state as a tuple.

As suggested in the definition, we usually write $\mathcal{I}d$ for $\mathcal{I}d(\Gamma)$ and **State** for **State**(Γ) except where omitting Γ may lead to confusion.

(Sometimes a collection of functions of this kind, with local restrictions on where each output can live, is called a display space. Each such function is a choice function, since when supplied with an input it chooses an element from the component with that index.)

The language *While* has three syntactic categories; identifiers X , expressions A , and instructions P . We are going to assign a meaning

$$\llbracket X \rrbracket \quad \llbracket A \rrbracket \quad \llbracket P \rrbracket$$

to each of these.

(Here ‘ $\llbracket \cdot \rrbracket$ ’ are the **semantic brackets** and are often used to indicate a process that attaches a meaning to a piece of syntax. In most parts of mathematics it is not too important to distinguish between syntax and meaning, but sometimes it is. For instance, the completeness theorem for predicate calculus is concerned with this distinction. The study of group presentations, as opposed to groups, is another place where it is useful to think in terms of syntax rather than semantics.)

Each of $\llbracket X \rrbracket$, $\llbracket A \rrbracket$, $\llbracket P \rrbracket$ is a function which consumes a state σ and returns a value $\llbracket X \rrbracket \sigma$, $\llbracket A \rrbracket \sigma$, $\llbracket P \rrbracket \sigma$ of an appropriate kind. The idea behind the first two is obvious, but the construction of $\llbracket P \rrbracket \sigma$ will take a bit more time. Let’s deal with the first two quickly.

Given an identifier X and a state σ the intention is that $\llbracket X \rrbracket \sigma$ is the value assigned to X by σ . Thus

$$\llbracket X \rrbracket \sigma = \sigma X$$

is the formal definition. Notice that this gives a function

$$\text{State} \xrightarrow{\llbracket X \rrbracket} \mathbb{X}_\perp$$

where \mathbb{X} is the housing type of X in the global context.

This looks a little pointless, and on its own it is. It is done here because it is the first step in a series of constructions which have a similar form and lead to a uniform treatment of the whole process.

Each expression A of the language is built up in a way to indicate how (the values attached to) the identifiers occurring in A can be combined to produce a compound value. The combinations allowed are determined by the operations built into the language, and are usually of a quite simple nature. The intention is that each expression A gives a function

$$\text{State} \xrightarrow{\llbracket A \rrbracket} \mathbb{A}_\perp$$

where \mathbb{A} is one of the housing types in the context. Thus $\llbracket A \rrbracket$ is that function which, when supplied with a state σ (that is with values for the identifiers) will return a compound value $\llbracket A \rrbracket \sigma$.

The function $\llbracket A \rrbracket$ is built up by recursion on the construction of A . We need not go into the details but an example will help.

3.2 EXAMPLE. Suppose the language has a binary operation symbol \boxplus which is allowed in expression. Thus if B, C are expressions then

$$(B \boxplus C)$$

is an expression. This primitive piece of syntax \boxplus will be the name of some concrete operation \odot . The meaning of $(B \boxplus C)$ is defined by

$$\llbracket (B \boxplus C) \rrbracket \sigma = (\llbracket B \rrbracket \sigma \odot \llbracket C \rrbracket \sigma)$$

for each state σ .

In other words, to evaluate the operation (named by) $(B \boxplus C)$ at the inputs given by σ we first evaluate the two components B and C at σ and then combine these values using the operation \odot indicated by the symbol \boxplus . Which is just what we've been doing all our lives. ■

This example illustrates that if the expressions are built in accordance with the typing discipline, then the semantics will produce a function of the correct type.

The example doesn't quite illustrate how smooth the whole process is. However, this will become more apparent in Definition 3.4, and is something you should look out for. It seems that all we are doing is combining functions in certain ways. This clarity arises because of the slightly different view of a

state we introduced in Definition 3.1. This kind of neatness is often referred to as ‘compositionality’ but rarely is there an explanation of what this notion is supposed to mean.

We come now to the meaning $\llbracket P \rrbracket$ of an instruction P . This again is a function, but this time from **State** to **State**. In other words $\llbracket P \rrbracket$ is a state transformer. We will take some time with the construction and the analysis of its properties.

Since each instruction P is built by recursion over syntax, the only sensible way to construct the function $\llbracket P \rrbracket$ to track that recursion. In particular, the meaning of assignments

$$\llbracket X := A \rrbracket$$

will be the base of the tracked recursion.

Think of how an assignment

$$X := A$$

can change the state. Almost all the registers are left unchanged. Only the contents of X can change, for the previous contents are replaced by the value of A . We mimic this kind of change on states.

3.3 DEFINITION. Let σ be a state over some context Γ , let $X \in \mathcal{Id}$ and let $u \in \mathbb{X}(X)_\perp$. The **update**

$$\sigma[X \mapsto u]$$

is the state give by

$$\sigma[X \mapsto u]Y = \begin{cases} \sigma X & \text{if } Y \text{ is not } X \\ u & \text{if } Y \text{ is } X \end{cases}$$

for each $Y \in \mathcal{Id}$. ■

In other words $\sigma[X \mapsto u]$ is the function which behaves like σ except that for input X it takes the new value u . This is the analogue of replacing the value held in X by u .

3.4 DEFINITION. For each program $\mathcal{P} = \Gamma | P$ the state transformer

$$\text{State} \xrightarrow{\llbracket P \rrbracket} \text{State}$$

is generated recursively by the clauses of Table 1 (for an arbitrary state σ). ■

These clauses need to be explained. The first two are straight forward, and the last is immediate, but the third (for the while loop) will take a bit of time to unravel. Let’s deal with the routine stuff first.

The behaviour of an update is precisely that required by an assignment. In fact, that is why we introduced the notion of an update.

if P is $X := A$	then $\llbracket P \rrbracket \sigma = \sigma[X \mapsto u]$	where $u = \llbracket A \rrbracket \sigma$
if P is $Q; R$	then $\llbracket P \rrbracket \sigma = \llbracket R \rrbracket (\llbracket Q \rrbracket \sigma)$	
if P is $\begin{array}{l} \textit{while } B \textit{ do} \\ Q \\ \textit{od} \end{array}$	then $\llbracket P \rrbracket \sigma = \begin{cases} \llbracket P \rrbracket \tau & \text{if } v \text{ is true} \\ \sigma & \text{if } v \text{ is false} \\ \perp & \text{otherwise} \end{cases}$	where $\begin{array}{l} \tau = \llbracket Q \rrbracket \sigma \\ v = \llbracket B \rrbracket \sigma \end{array}$
if P is $\begin{array}{l} \textit{if } B \textit{ then do} \\ Q \\ \textit{elsedo} \\ R \\ \textit{fi} \end{array}$	then $\llbracket P \rrbracket \sigma = \begin{cases} \llbracket Q \rrbracket \sigma & \text{if } v \text{ is true} \\ \llbracket R \rrbracket \sigma & \text{if } v \text{ is false} \\ \perp & \text{otherwise} \end{cases}$	where $v = \llbracket B \rrbracket \sigma$

Table 1: The recursion clauses for the meaning of instructions

What happens when a sequential composite

$$Q; R$$

is performed? Starting with the current state we perform Q and then, using the modified state, we perform R . This is precisely what the second clause says. We have

$$\llbracket Q; R \rrbracket = \llbracket R \rrbracket \circ \llbracket Q \rrbracket$$

in terms of function composition. Notice how the use of the two components have to be in the correct order.

(A word of warning here. Some people think it is better to write the composite $g \circ f$ of two functions as $f; g$, under the misguided belief that this will simplify various calculations. It doesn't. Usually a function receives its input from the right. Thus

$$(g \circ f)x = g(fx)$$

for each input x . Now look what happens when we use the 'better' notation. We have

$$(f; g)x = g(fx)$$

for each input x , which is a mess. There are some subjects, such as the study of modules over a ring, where both left- and right-actions are used. In such cases some functions do receive inputs from the left and then, and only then, does a different notation for composition become sensible. There is, of course, one common function which always receives its inputs from the left.)

The intuitive meaning of the conditional clause is obvious. Using the current state we evaluate the boolean condition B then, depending on the outcome,

we carry out either Q or R . This is precisely what the conditional clause of Definition 3.4 says.

We come now to the analysis of the while loop. This is the main aim of this section. To do this and to help with later developments, it is convenient to introduce some notation. Thus, for instructions P, Q, R, \dots we write

$$\mathcal{P} = \llbracket P \rrbracket \quad \mathcal{Q} = \llbracket Q \rrbracket \quad \mathcal{R} = \llbracket R \rrbracket \quad \dots$$

for the corresponding state transformer.

Consider the while loop P

while B do Q od

built from an expression B and an instruction Q . The corresponding function $\mathcal{P} = \llbracket P \rrbracket$ is determined by the value $\llbracket B \rrbracket$ and the function $\mathcal{Q} = \llbracket Q \rrbracket$. Think of how we must carry out the instruction P starting from a state σ .

We first evaluate B to obtain a value $v = \llbracket B \rrbracket \sigma$. For the loop to make sense this should be either true or false. If it turns out to be something else then either the instruction is incorrectly written, or it is being used in the wrong way. In this case we agree to output \perp (which we can think of as ‘error’).

Suppose v is a boolean value.

If v is false then there is nothing to do and the state is unchanged. Thus

$$\mathcal{P}\sigma = \sigma$$

in this case.

If v is true then we execute the instruction Q just once and, using the new state, we return to the beginning of the loop. The new state is $\tau = \mathcal{Q}\sigma$ and so

$$\mathcal{P}\sigma = \mathcal{P}\tau$$

is the effect of running the loop.

This may look circular. It is not, but it is a form of recursion. A run of the while loop will repeatedly pass through the body of the loop (performing that instruction at each pass) until the boolean condition B becomes false. Remember that the value of B may depend on the current value of certain identifiers, and these can change on each pass through the body.

We hope the body of the loop will be executed just finitely many times. Suppose this is so and, there are r passes through the loop. Then

$$\mathcal{P}\sigma = \mathcal{Q}^r \sigma$$

where σ is the state before the loop is entered. Of course the number of passes depends on σ , so we can not fix r beforehand.

In some circumstances the boolean condition B may never become false. In this case the run never leaves the loop, and keeps on passing through the

body forever. When this happens we agree to a nominal output \perp (which we can think of as ‘non-termination’ or ‘undefined’). In general, there is no way of building into a while loop a check for non-termination. This is because there are recursively enumerable sets that are not recursive.

3.5 EXAMPLE. Consider a cleaner version of the program MLT of Examples 2.2. This uses the while loop P

$$\mathit{while} \ X \neq 0 \ Z := Y + Z; X := X - 1 \ \mathit{od}$$

which is run in the context

$$X : \mathbb{N}, Y : \mathbb{Z}, Z : \mathbb{Z}$$

(since the identifier I is not needed).

Let Q be the body of the loop, that is a sequential composite of two assignments. Consider a state σ with

$$\sigma X = x \quad \sigma Y = y \quad \sigma Z = z$$

where $x \in \mathbb{N}$ and $y, z \in \mathbb{Z}$.

The first assignment of Q gives an update

$$\llbracket Z := Y + Z \rrbracket \sigma = \sigma[Z \mapsto y + z]$$

(since $\llbracket X \rrbracket \sigma = x$, $\llbracket Y \rrbracket \sigma = y$, $\llbracket Z \rrbracket \sigma = z$). This is followed by a second assignment to give

$$\llbracket Q \rrbracket \sigma = \llbracket X := X - 1 \rrbracket (\sigma[Z \mapsto y + z])[X \mapsto u - 1]$$

where

$$u = \sigma[Z \mapsto y + z]X = \sigma X = x$$

gives the second update.

This is a rather laborious way of going through a calculation which, in this case, is almost trivial. As earlier we can write down a state as a triple

$$\sigma = (x, y, z)$$

by listing the three component values. We can think of the meaning $\llbracket R \rrbracket$ of an instruction R as a transition

$$(x, y, z) \xrightarrow{R} (x^*, y^*, z^*)$$

where

$$x^* = (\llbracket R \rrbracket(x, y, z))X \quad y^* = (\llbracket R \rrbracket(x, y, z))Y \quad z^* = (\llbracket R \rrbracket(x, y, z))Z$$

are the new components. In this notation we see that

$$(x, y, z) \xrightarrow{Z := Y + Z} (x, y, y + z) \xrightarrow{X := X - 1} (x - 1, y, y + z)$$

are the two components of Q to describe the overall transition effect.

The while loop now repeats this transition until the first component of the state becomes 0. Thus we have

$$\begin{array}{l}
 (x, y, z) \xrightarrow{Q} (x - 1, y, y + z) \\
 \xrightarrow{Q} (x - 2, y, 2y + z) \\
 \xrightarrow{Q} (x - 3, y, 3y + z) \\
 \xrightarrow{Q} (x - 4, y, 4y + z) \\
 \xrightarrow{Q} \dots
 \end{array}$$

which could continue for some time. It is routine to see that

$$(x, y, z) \xrightarrow{Q^i} (x - i, y, iy + z)$$

where Q^i indicates i applications of Q . Thus

$$(x, y, z) \xrightarrow{P} (0, y, xy + z)$$

is the effect of this while loop (provided the input x is positive as requested by the context). ■

In practice we don't need to be so detailed to see what is happening. Often we can display the behaviour pictorially. Thus

X	Y	Z
x	y	z
$x - 1$	y	$y + z$
$x - 2$	y	$2y + z$
$x - 3$	y	$3y + z$
$x - 4$	y	$4y + z$
\vdots	\vdots	\vdots

shows us the state changes on each pass through the loop.

The construction of the transition function of a program seems to follow the intended idea behind the various program language constructs. Does this suggest that verifying a program against its specification is easy? Think about this point. By tracking through the syntax of a program we obtain a recursive specification of its meaning function. This must have some relationship to the specification of the function the program is supposed to evaluate. Perhaps they are essentially the same specification? Look at the clause in Table 1 giving the meaning $\llbracket P \rrbracket$ of a while loop P . This is almost a tail recursion. (In fact, it is precisely a tail recursion in a more general setting.) However, suppose this instruction P has been obtained from a specification that is not a tail recursion. (For instance, we might start with head recursion.) In such a case something

has happened in the programming process. What is this? Don't hold your breath.

To conclude this section we give a brief mention of another loop construct. This is similar to the while loop but is much, much weaker. The syntax of the for loop is

For A do Q od

where A is an expression and Q is an instruction. To carry out this compound instruction in some state σ the machine first calculates $m = \llbracket A \rrbracket \sigma$ (which must be a natural number) and then performs Q exactly m times. The crucial difference is that here the bound m is known before the body of the loop is entered, whereas in a while loop the bound depends on the effect of the loop (and may not exist). This is similar to the difference between bounded search and unbounded search discussed in *Forms of recursion* (although that analogy is slightly misleading).

If we modify *While* by replacing the while loop by the for loop, then we obtain a much weaker language. As mentioned earlier, *While* is Turing complete, that is for each general recursive function there is at least one *While* program that computes that program. However, in the modified language only the primitive recursive functions can be programmed.

(Strictly speaking these last remarks are correct only when we deal with first order numeric functions. It is in this setting that these notes are written. However, it is perfectly possible to program certain higher order functions, such as the Ackermann jump, provided we can store lists of variable length. With these and for loops we can get beyond primitive recursion.

Exercises

6 The standard recursive specification of multiplication over \mathbb{N} is

$$M(0, y) = 0 \quad M(x', y) = y + M(x, y)$$

(for $x, y \in \mathbb{N}$). This calls on addition which we assume is built into the language.

(a) Convert the head recursive specification of M into an initialize while loop, **IT;LP**. (This needs an extra register to count up to the recursion argument.)

(b) Write down the tail recursive specification of **LP**.

(c) (Can you set up an induction to prove that **LP** give the correct result (when it is supposed to)?)

(d) Compare the initial and final specifications. Can you think of a better way of programming multiplication?

7 Consider the uninitialized while loop of each of the three programs of Exercise 2 (that is, without the initial assignment to Z). The transition function of each is

$$(x, y, z) \longmapsto (0, y^{2^x}, zy^x)$$

where e depends on x and the particular loop. (For the left-hand loop $e = 0$.)

(a) By analysing the construction of the meaning function, provide a formal proof of the claim above.

(b) Can you make any observations about the complexity of each loop (in terms of the number of steps it takes to terminate) and the complexity of the formal verification?

8 In the following σ is a state, X, Y are distinct identifiers (in the support of σ) and u, v are values. For each of the updated states δ , give an explicit description of δX and δY in terms of σ .

- | | |
|--|--|
| (a) $\sigma[X \mapsto u][Y \mapsto \sigma X][X \mapsto v]$ | (b) $\sigma[X \mapsto u][Y \mapsto v][X \mapsto \sigma Y]$ |
| (c) $\sigma[X \mapsto \sigma Y][Y \mapsto \sigma X]$ | (d) $\sigma[X \mapsto \sigma[Y \mapsto v]Y]$ |
| (e) $\sigma[X \mapsto \sigma[Y \mapsto \sigma X]Y]$ | (f) $\sigma[X \mapsto \sigma[Y \mapsto \sigma X]X]$ |

9 Two instructions P, Q (in the same context) are said to be **denotationally equivalent** if $\llbracket P \rrbracket = \llbracket Q \rrbracket$. Determine which of the following pairs are equivalent, and in each case justify your claim.

- | | |
|---|--|
| (i) $(P; Q); R$ | $P; (Q; R)$ |
| (ii) <i>if B then do Q else do Q fi</i> | Q |
| (iii) $P; \textit{if B then do Q else do R fi}$ | <i>if B then do P; Q else do P; Q fi</i> |
| (iv) <i>if B then do P else do Q fi; R</i> | <i>if B then do P; Q; R else do Q; R fi</i> |
| (v) <i>while B do Q od; R</i> | <i>if B then do Q; P; R else do R fi</i>
where P is the left-hand instruction |

- | | |
|---|---------------------------------------|
| (vi) <i>if B then do</i>
<i>if B then do P else do Q fi</i>
<i>else do R</i>
<i>fi</i> | <i>if B then do P else do Q; R fi</i> |
|---|---------------------------------------|

- | | |
|--|---|
| (vii) <i>if A then do</i>
<i>if B then do P else do Q fi</i>
<i>else do</i>
<i>if B then do P else do R fi</i>
<i>fi</i> | <i>if B then do</i>
P
<i>else do</i>
<i>if A then do Q else do R fi</i>
<i>fi</i> |
|--|---|

10 (a) Show that if the function

$$\mathbb{N}, \mathbb{P} \xrightarrow{h} \mathbb{T}$$

can be programmed in *While* then so can the function $(\mu y)[h(y, \cdot) = 0]$.

(b) Show that the transition function of the for loop

$$\textit{For A do Q od}$$

is ‘primitive recursive’ in the transition function of Q and the meaning of A .

(c) Why ‘primitive recursive’?

(d) Explain why the for loop is stronger than bounded search.

4 Head and tail recursion

In this section we see how a head or tail recursion over \mathbb{N} can be programmed, and we compare the original specification of the function with that of the transition function of the program (generated via the denotational semantics). In the next section we will look at some more refined recursions.²

We begin with the easiest case, tail recursion.

There are two variant of a simple tail recursion over \mathbb{N} . Using the data functions g, k , as to the left,

$$\mathbb{P} \xrightarrow{g} \mathbb{T} \quad \mathbb{N}, \mathbb{P} \xrightarrow{k} \mathbb{P} \quad \mathbb{N}, \mathbb{P} \xrightarrow{f} \mathbb{T}$$

we specify the function f , as to the right, by either of

$$\begin{array}{l} f(0, p) = gp \\ f(x', p) = f(x, p^+) \\ \text{where } p^+ = k(x, p) \end{array} \quad f(x, p) = \begin{cases} gp & \text{if } x = 0 \\ f(x^-, p^+) & \text{if } x \neq 0 \\ \text{where } x^- = x - 1 \\ \quad p^+ = k(x, p) \end{cases}$$

for $x \in \mathbb{N}, p \in \mathbb{P}$. Of course, for a given k these are not the same function, but each can be rephrased as the other using a slight variant of k .

The program for each of these is almost identical with the specification.

$$\begin{array}{c} \frac{\text{TAIL}[X : \mathbb{N}, P : \mathbb{P}, T : \mathbb{T}]}{\text{while } X \neq 0 \text{ do} \\ \quad X := X - 1; \\ \quad P := k(X, P) \\ \text{od}; \\ T := g(P)} \end{array} \quad \begin{array}{c} \frac{\text{TAIL}[X : \mathbb{N}, P : \mathbb{P}, T : \mathbb{T}]}{\text{while } X \neq 0 \text{ do} \\ \quad P := k(X, P); \\ \quad X := X - 1 \\ \text{od}; \\ T := g(P)} \end{array}$$

Observe that the only difference between these two program is the order of the two assignments in the body of the loop.

Before we compare the semantic function of each program with its parent specification, it is useful to uncouple the final assignment from the loop. Thus, using the data function k consider the function

$$\mathbb{N}, \mathbb{P}, \mathbb{T} \xrightarrow{F} \mathbb{T}$$

specified by

$$\begin{array}{l} F(0, p, t) = t \\ F(x', p, t) = F(x, p^+, t) \\ \text{where } p^+ = k(x, p) \end{array} \quad F(x, p, t) = \begin{cases} t & \text{if } x = 0 \\ F(x^-, p^+, t) & \text{if } x \neq 0 \\ \text{where } x^- = x - 1 \\ \quad p^+ = k(x, p) \end{cases}$$

²In fact, as explained in section 1, the next section is merely a draft of the material to be covered.

respectively, where $x \in \mathbb{N}, p \in \mathbb{P}, t \in \mathbb{T}$. Thus

$$f(x, p) = F(x, p, gp)$$

holds by a simple induction over x .

Let

$$\mathbb{N} \times \mathbb{P} \times \mathbb{T} \xrightarrow{\mathcal{T}} \mathbb{N} \times \mathbb{P} \times \mathbb{T}$$

be the semantic function of the loop instruction in either program. (Strictly speaking the type of \mathcal{T} is not quite correct, for there should be some ‘error’ values \perp involved.) By definition, we have

$$\mathcal{T}(x, p, t) = \begin{cases} (x, p, t) & \text{if } x = 0 \\ \mathcal{T}(x^-, p^+, t) & \text{if } x \neq 0 \\ \text{where } x^- = x - 1 \\ p^+ = k(x^-, p) \end{cases} \quad \mathcal{T}(x, p, t) = \begin{cases} (x, p, t) & \text{if } x = 0 \\ \mathcal{T}(x^-, p^+, t) & \text{if } x \neq 0 \\ \text{where } x^- = x - 1 \\ p^+ = k(x, p) \end{cases}$$

for the two cases. Make sure you spot the difference between these. The transition function for the whole program (with the final assignment) is

$$(x, p, \cdot) \longmapsto \mathcal{T}(x, p, gp)$$

for $x \in \mathbb{N}, p \in \mathbb{P}$.

Another routine induction over x gives

$$\mathcal{T}(x, p, t) = F(x, p, t)$$

and hence

$$\mathcal{T}(x, p, gp) = F(x, p, gp) = f(x, p)$$

to show that the program does produce the value of f .

This is a simple example of a **verification** of a program against a specification. That is, it is a proof that the program can be used to evaluate a function which meets the specification.

Of course, not all verifications are this trivial. The proof here is easy because tail recursion is tailor made for programming (which might be where the word comes from). There is hardly any difference between the parent specification, the program, and the resulting transition function. This is not the case even for some quite simple specifications.

To see this let’s turn to the analysis of a head recursion.

As usual there are minor variants of head recursion. Here we will look at the b/s version (and in the next section we will consider how much more general forms can be handle).

Using data functions g, h , as to the left,

$$\mathbb{P} \xrightarrow{g} \mathbb{T} \quad \mathbb{N}, \mathbb{P}, \mathbb{T} \xrightarrow{h} \mathbb{P} \quad \mathbb{N}, \mathbb{P} \xrightarrow{f} \mathbb{T}$$

we specify the function f , as to the right, by

$$f(0, p) = gp \quad f(x', p) = h(x, p, t) \text{ where } t = f(x, p)$$

for $x \in \mathbb{N}, p \in \mathbb{P}$.

When first seen this causes a minor problem. To calculate, say $f(7, p)$ we must first calculate $t = f(6, p)$ and then substitute this value into $h(7, p, t)$. Before we can do that we need $f(6, p)$, so we must first calculate $t = f(5, p)$ and then substitute this value into $h(6, p, t)$ which can then be substituted into $h(7, p, \cdot)$. We soon see the trick. We calculate

$$f(0, p), f(1, p), f(2, p), \dots$$

in turn until we get to the required value.

This is easy to program. All we need is an extra register to count up to the required value. It doesn't take too long to see that

$$\frac{\text{HEAD}[I, X : \mathbb{N}, P : \mathbb{P}, T : \mathbb{T}]}{T := gP; I := 0; \\ \textit{while } I < X \textit{ do} \\ \quad T := h(I, P, T); \\ \quad I := I + 1 \\ \textit{od}}$$

will do the job, and even better it is not hard to prove that it does. However, it is worth trying to set the proof in a slightly more general context.

As with tail recursion, to analyse the meaning of the program it is convenient to uncouple the loop from the initialization assignments.

Using the data function h let

$$\mathbb{N}, \mathbb{P}, \mathbb{T} \xrightarrow{F} \mathbb{T}$$

be the function specified by

$$F(0, p, t) = t \quad F(x', p, t) = F(x, p, t^+) \text{ where } t^+ = F(x, p, t)$$

(for $x \in \mathbb{N}, p \in \mathbb{P}, t \in \mathbb{T}$). Notice that this is a tail recursive specification. It doesn't take too long to see that

$$f(x, p) = F(x, p, gp)$$

and hence the specification of F is the essential content of that of f . In particular, if we can program F then we can program f by initializing the input t . This is how the program HEAD arises.

Let

$$\mathbb{N} \times \mathbb{P} \times \mathbb{T} \xrightarrow{\mathcal{H}} \mathbb{N} \times \mathbb{P} \times \mathbb{T}$$

be the semantic function of the loop instruction in the program. (As with \mathcal{T} above, the source and target of \mathcal{H} should be lifted.) By definition, we have

$$\mathcal{H}(i, x, p, t) = \begin{cases} (i, x, p, t) & \text{if } x \leq i \\ \mathcal{H}(i', x, p, h(i, p, t)) & \text{if } i < x \end{cases}$$

for $i, x \in \mathbb{N}, p \in \mathbb{P}, t \in \mathbb{T}$. Using \mathcal{H} we see that the meaning \mathcal{F} of HEAD is

$$(\cdot, x, p, \cdot) \longmapsto \mathcal{H}(0, x, p, gp)$$

and we begin to see how we must relate F and \mathcal{H} . However, the trick we use is not quite so obvious. The following is proved by a suitable induction (which you should think about).

4.1 LEMMA. *For the function F and related transition function \mathcal{H} we have*

$$\mathcal{H}(i, x + i, p, F(i, p, t)) = (x + i, x + i, p, F(x + i, p, t))$$

for each $i, x \in \mathbb{N}, p \in \mathbb{P}, t \in \mathbb{T}$.

Given this result we may set $i = 0$ to get

$$\mathcal{H}(0, x, p, t) = (x, x, p, F(x, p, t))$$

(since $H(0, p, t) = t$), and hence the behaviour of the program Head is

$$(\cdot, x, p, \cdot) \longmapsto \mathcal{H}(0, x, p, gp) = (x, x, p, H(x, p, gp)) = (x, x, p, f(x, p))$$

which gives a verification of the program.

This simple example shows that we can make life easier if we do a little work before we program. If we can rephrase the parent specification to look more like a tail recursion, then the resulting specification will be easier to program. However, such a rephrasing is not always a straight forward exercise.

There are various tricks that can be used in appropriate situations. Some of these are dealt with in the Exercises.

Exercises

11 Prove Lemma 4.1.

12 Sometimes a recursion can be made more tail-like by the introduction of an **accumulator**, that is an extra argument whose job is to hold some temporary values.

Consider the 3-placed function over the \mathbb{N} specified by

$$M(0, y, z) = z \quad M(x', y, z) = M(x, y, y + z)$$

(for $x, y, z \in \mathbb{N}$).

- (a) Give an explicit definition of this function.
- (b) Convert the specification into a while loop and write down the meaning function of this.
- (c) Compare this result with your answer to Exercise 6.

13 Using a function h of a suitable type, consider the functions H, T specified by

$$\begin{array}{ll} H(0, p, t) = t & T(0, p, t) = t \\ H(x', p, t) = h(p, t^+) & T(x', p, t) = T(x, p, t^-) \\ \text{where } t^+ = H(x, p, t) & \text{where } t^- = h(p, t) \end{array}$$

where $x \in \mathbb{N}$ and p, t are parameters.

- (a) Write down the types of h, H, T .
- (b) Show that

$$T(x, p, H(y, p, t)) = H(x + y, p, t)$$

holds for all $x, y \in \mathbb{N}$ and parameters p, t .

- (c) Show that $H = T$.
- (d) Can you find a better explanation of the equality of (b)?

14 Using a function h of a suitable type, consider the functions H, T specified by

$$\begin{array}{ll} H(0, p, t) = t & T(0, p, t) = t \\ H(x', p, t) = h(x, p, t^+) & T(x', p, t) = T(x, p, t^-) \\ \text{where } t^+ = H(x, p, t) & \text{where } t^- = h(x, p, t) \end{array}$$

where $x \in \mathbb{N}$ and p, t are parameters. Suppose the function h has the exchange property

$$h(u, p, h(v, p, t)) = h(v, p, h(u, p, t))$$

for all $u, v \in \mathbb{N}$ and parameters p, t .

- (a) Write down the types of h, H, T .
- (b) Show that

$$H(x, p, h(i, p, t)) = h(i, p, H(x, p, t)) \quad T(x, p, h(i, p, t)) = h(i, p, T(x, p, t))$$

hold for all $x, i \in \mathbb{N}$ and parameters p, t .

- (c) Show that $H = T$.
- (d) Can you find a better explanation of the equality of (b)?

[The original intention was to include the material of the next section in the notes and accompanying lectures. This was not done because of lack of time. However, the first draft of that section is included in these notes.]

5 More refined recursions

In the previous section we saw that a tail recursive specification is very easy to program and verify, and a head recursive specification is reasonably easy to deal with. To be precise, we saw how to handle a b/s head recursion over \mathbb{N} . But what about other forms of head recursions? In this section we first look at a slightly more refined form of head recursion, and in doing so we see how to deal with some quite general recursions.

In a b/s recursion over \mathbb{N} the recursion variable decreases by 1 at each pass through the recursion, and the recursion bottoms out when this variable reaches 0. There are many recursions, even over \mathbb{N} , which do not operate in this way. For instance, we could replace the standard recursion step by

$$f(x, p) = h(x, p, t) \text{ where } t = f(x^-, p), x^- = dx$$

where $d \in \mathbb{N}'$ is a ‘descent’ function. The b/s version is essentially the case where $dx = x - 1$ (after a slight reorganization of the data function h).

To evaluate a function f using a b/s head recursion we calculate

$$f(0, p), f(1, p), \dots, f(i, p), \dots, f(x, p)$$

in turn. Furthermore, the evaluation of $f(i', p)$ requires only $f(i, p)$ of the previous values, so once used each previous value can be destroyed.

A descent recursion is not so simple. Certainly the evaluation of $f(x, p)$ uses just one ‘earlier’ value, but we don’t know which one until we get into the calculation. We could evaluate and store all ‘earlier’ values, and use these as they are needed. Even if we do this we can’t destroy a value as it is used, for it might be needed again later. (There are some quite weird descent functions.)

A better way is not to work from the bottom up, but from the top down. After a bit of thought we see that we can evaluate $f(x, p)$ as follows.

- Starting with the input x we use the descent function d to calculate and store

$$x(0) = x, x(1) = dx(0), \dots, x(i') = dx(i), \dots$$

where this sequence continues until it bottoms out at $x(l)$ for some index l . We thus have stored the list

$$[x(l), x(l-1), \dots, x(i), \dots, x(0)]$$

of recursion inputs needed to evaluate $f(x, p)$.

- Using this store list we evaluate

$$\begin{aligned}
t(l) &= f(x(l), p) = \dots \\
t(l-1) &= f(x(l-1), p) = h(x(l-1), p, t(l)) \\
&\vdots \\
t(i) &= f(x(i), p) = h(x(i), p, t(i+1)) \\
&\vdots \\
t(0) &= f(x(0), p) = h(x(0), p, t(1))
\end{aligned}$$

in turn to obtain the required $f(x, p)$.

Notice that once used the head of the stored list can be removed and destroyed. Notice also that only one earlier value of the function needs to be stored.

Once we have seen this trick we realize that it will work in more general situations.

5.1 EXAMPLE. Using the data functions

$$\mathbb{X} \xrightarrow{g} \mathbb{T} \qquad \mathbb{X}, \mathbb{P}, \mathbb{T} \xrightarrow{h} \mathbb{T} \qquad \mathbb{X}, \mathbb{P} \xrightarrow{d} \mathbb{X}$$

and a subset $\mathbb{B} \subseteq \mathbb{X}$, let

$$\mathbb{X}, \mathbb{P} \xrightarrow{f} \mathbb{T}$$

be the function specified by

$$f(x, p) = \begin{cases} g(x, p) & \text{if } x \in \mathbb{B} \\ h(x, p, t) & \text{if } x \notin \mathbb{B} \end{cases} \text{ where } \begin{cases} t = f(x^-, p) \\ x^- = d(x, p) \end{cases}$$

for $x \in \mathbb{X}, p \in \mathbb{P}$. ■

This example is slightly more general than before. The most obvious modification is that the change $x \mapsto x^-$ in the recursion variable now depends of the parameter p as well. Notice also that the domain \mathbb{X} of the recursion variable x need not be the natural numbers, and we don't need x^- to be 'smaller' than x . To take care of that we use a subset $\mathbb{B} \subseteq \mathbb{X}$ which is where the recursion bottoms out.

More or less the same format as before can be used to evaluate f .

- Starting with the inputs x, p we use the function d to calculate and store

$$x(0) = x, x(1) = d(x(0), p), \dots, x(i') = d(x(i), p), \dots$$

where this sequence continues until we find the least index l with $x(l) \in \mathbb{B}$.

- After this first phase we track back through the stored list to calculate

$$\begin{aligned}
t(l) &= f(x(l), p) = g(x(l), p) \\
&\quad \vdots \\
t(i) &= f(x(i), p) = h(x(i), p, t(i+1)) \\
&\quad \vdots \\
t(0) &= f(x(0), p) = h(x(0), p, t(1))
\end{aligned}$$

in turn to obtain the required $f(x, p)$.

Of course, in general there is no reason why the first phase should ever terminate, that is to find some $x(l) \in \mathbb{B}$. This means that the specified function f is partial. However, this is not a problem, for three reasons. Firstly, if we set up the recursion (rather than obtain it from elsewhere) then we know about the potential partiality and only use the function where it is defined. Secondly, when writing the program we will know about the partiality and, if necessary, take the appropriate precautions. Thirdly, the topic of **domain theory** is designed to handle this kind of situation (which it does without effort).

Almost the same method handles body recursion.

5.2 EXAMPLE. Using the data functions

$$\mathbb{X} \xrightarrow{g} \mathbb{T} \quad \mathbb{X}, \mathbb{P}, \mathbb{T} \xrightarrow{h} \mathbb{T} \quad \mathbb{X}, \mathbb{P} \xrightarrow{d} \mathbb{X} \quad \mathbb{X}, \mathbb{P} \xrightarrow{e} \mathbb{P}$$

and a subset $\mathbb{B} \subseteq \mathbb{X}$, let

$$\mathbb{X}, \mathbb{P} \xrightarrow{d} \mathbb{T}$$

be the function specified by

$$f(x, p) = \begin{cases} g(x, p) & \text{if } x \in \mathbb{B} \\ h(x, p, t) & \text{if } x \notin \mathbb{B} \text{ where } \begin{cases} t = f(x^-, p^+) \\ x^- = d(x, p) \\ p^+ = e(x, p) \end{cases} \end{cases}$$

for $x \in \mathbb{X}, p \in \mathbb{P}$. ■

In this recursion both the recursion variable and the parameter change on each pass across the recursion. This means that we don't know which parameter will be needed when the recursion bottoms out. However, this is hardly a problem.

We can evaluate f as follows.

- Starting with the inputs x, p we generate and store two lists

$$\begin{aligned}
x(0) &= x, x(1) = d(x(0), p(0)), \dots, x(i') = d(x(i), p(i)), \dots \\
p(0) &= p, p(1) = e(x(0), p(0)), \dots, p(i') = e(x(i), p(i)), \dots
\end{aligned}$$

where this sequence continues until we find the least index l with $x(l) \in \mathbb{B}$.

- After this first phase we track back through the stored lists to calculate

$$\begin{aligned}
 t(l) &= f(x(l), p(l)) = g(x(l), p(l)) \\
 &\quad \vdots \\
 t(i) &= f(x(i), p(i)) = h(x(i), p(i), t(i+1)) \\
 &\quad \vdots \\
 t(0) &= f(x(0), p(0)) = h(x(0), p(0), t(1))
 \end{aligned}$$

in turn to obtain the required $f(x, p)$.

The extra cost here is that we must now store a pair of lists (rather than just one list). However, this hardly make the analysis more complicated. In fact, it leads to a simplification.

We must generate and store a pair of lists

$$\begin{aligned}
 &[x(l), \dots, x(i), \dots, x(0)] \\
 &[p(l), \dots, p(i), \dots, p(0)]
 \end{aligned}$$

of equal length. But a pair of list is, more or less, the same as a list

$$[(x(l), p(l)), \dots, (x(i), p(i)), \dots, (x(0), p(0))]$$

of pairs. using this idea we can re-view Example 5.2 using a pair

$$s = (x, p)$$

as the recursion variable.

5.3 EXAMPLE. Using the data functions

$$\mathbb{S} \xrightarrow{g} \mathbb{T} \qquad \mathbb{S}, \mathbb{T} \xrightarrow{h} \mathbb{T} \qquad \mathbb{S} \xrightarrow{k} \mathbb{S}$$

and a subset $\mathbb{B} \subseteq \mathbb{X}$, let

$$\mathbb{S} \xrightarrow{f} \mathbb{T}$$

be the function specified by

$$f(x, p) = \begin{cases} gs & \text{if } s \in \mathbb{B} \\ h(s, t) & \text{if } s \notin \mathbb{B} \end{cases} \text{ where } \begin{cases} t = fs^- \\ s^- = ks \end{cases}$$

for $x \in \mathbb{X}, p \in \mathbb{P}$. ■

It is not too hard to rephrase the specification of Example 5.2 in this form. We take

$$\mathbb{S} = \mathbb{X} \times \mathbb{P}$$

and define k by

$$k(x, p) = (d(x, p), e(x, p))$$

for $(x, p) \in \mathbb{S}$. In fact, this specification is more general than that of Example 5.2, for now the bottoming out condition can depend on both x and p . (This recursion is exactly the one given in Example 3 of the notes ‘Forms of recursion and induction’.)

The way we evaluate this function is almost the same as the one we started with (which appeared to apply only to a descent head recursion over \mathbb{N}).

- Starting with the inputs s we generate and store a list

$$s(0) = s, s(1) = ks(0), \dots, s(i') = ks(i), \dots$$

where this sequence continues until we find the least index l with $s(l) \in \mathbb{B}$.

- After this first phase we track back through the stored list to calculate

$$\begin{aligned} t(l) &= fs(l) = gs(l) \\ &\quad \vdots \\ t(i) &= fs(i) = h(s(i), t(i+1)) \\ &\quad \vdots \\ t(0) &= fs(0) = h(s(0), t(1)) \end{aligned}$$

in turn to obtain the required fs .

It is now reasonably clear how we can program the rather general recursion of Example 5.3. We use two while loops in sequence with a joining assignment. The first loop generates and stores the list of values $s(i)$. The second loop uses the list to generate the values $t(i)$ (in reverse order of the index).

To implement this we need the facility to handle list of elements of \mathbb{S} .

***** It would be useful to do some arithmetic of lists. It highlights some things about recursion that are not clear with \mathbb{N} . However, I'm not sure there will be time.

If we can do some list then we can explain a bit more about the following program.

If we can't then we can still state it and let them think about it. *****

$$\frac{S : \mathbb{S}, T : \mathbb{T}, L : \mathbb{L} = \text{List}(\mathbb{S})}{L := [];$$

```

while  $\neg\beta(S)$  do
  L := S † L;
  S := k(S)
od;
T := g(S);
while L  $\neq$  [] do
  S := head(L);
  L := tail(L);
  T := h(S, T)
od

```

Exercises

- 15 *Look up course-of-values recursion and compare with descent recursion.*
- 16 *Need a few more.*

While loops – Some solutions

For section 1

1 (a) Since $[1] = 1[]$ the first few steps are

		0	0	[]	1[]
(1)	gives	1	1	[]	[1]
(2)	gives	1	0	[]	[1, 1]
(1)	gives	2	1	[1]	[1]
(3)	gives	2	1	[]	[2, 1]
(2)	gives	2	0	[]	[1, 2, 1]

and so on. After a while the transition path goes through a cycle

		5	0	[]	[1, 5, 10, 10, 5, 1]
(1)	gives	6	1	[5, 10, 10, 5, 1]	[1]
(3)	gives	6	5	[10, 10, 5, 1]	[6, 1]
(3)	gives	6	10	[10, 5, 1]	[15, 6, 1]
(3)	gives	6	10	[5, 1]	[20, 15, 6, 1]
(3)	gives	6	5	[1]	[15, 20, 15, 6, 1]
(3)	gives	6	1	[]	[6, 15, 20, 15, 6, 1]
(2)	gives	6	0	[]	[1, 6, 15, 20, 15, 6, 1]

and then continues in this fashion.

(b) In each state

$$(r, b, l, m)$$

the first component r is as counter and indicates which line of the triangle is being generated. To calculate $\binom{n}{i}$ run the transition until

$$(n, 0, [], m)$$

is achieved and then select the i^{th} component of the list m . ■

For section 2

2 The left hand program is easiest to understand. Starting with a 1 in Z it repeatedly multiplies the contents of X by the contents of Y and uses X to count the number of times this happens. Almost trivially

$$(x, y, \cdot) \longmapsto (0, y, y^x)$$

is the complete transition function. This takes x passes through the loop to achieve this.

The central program is a bit more sophisticated. On each pass through the loop it decides whether or not X holds an even number and then the state changes as

$$(2a, y, z) \mapsto (a, y^2, z) \quad (2a + 1, y, z) \mapsto (2a, y, z \times y)$$

accordingly. The value of

$$Y^X \times Z$$

is unchanged on each pass through the loop. On entering the loop the value of this $y^x \times 1$ and so $Y^X \times Z$ must still have this value on leaving the loop. But at that point X has value 0, and so Z must have value y^x . Thus

$$(x, y, \cdot) \mapsto (0, y^{2^e}, y^x)$$

is the complete transition function where e depends on x . This program is logarithmically faster (as measured by e).

The right hand program is similar to the central one, except this one gets rid of all powers of 2 in the value of X before it goes to the beginning of the outer loop. ■

3 Both the programs

$$\frac{[X, Y, Q, R : \mathbb{N}]}{R := X; Q := 0} \quad \frac{[X, Y, P, Q, R : \mathbb{N}]}{R := X; Q := 0}$$

$\begin{array}{l} \mathit{while} Y \leq R \mathit{ do} \\ \quad R := R - Y; \\ \quad Q := Q + 1 \\ \mathit{od} \end{array}$	$\begin{array}{l} \mathit{while} Y \leq R \mathit{ do} \\ \quad R := R - Y; \\ \quad Z := Y; \\ \quad P := 1; \\ \quad \mathit{while} 2Z \leq R \mathit{ do} \\ \quad \quad R := R - 2Z; \\ \quad \quad Z := 2Z; \\ \quad \quad P := 2P + 1 \\ \quad \mathit{od}; \\ \quad Q := Q + P; \\ \quad P := 0 \\ \mathit{od} \end{array}$
---	--

compute the two functions. The one on the right is quite a bit faster. To see this try the inputs $X = 500, Y = 7$. ■

4 (a) We first consider the restriction of f to $r \in \mathbb{N}$. Fix the parameters a, b, x, y and write f for $f(\cdot, a, b, x, y)$.

Let

$$\alpha^2 = a\alpha + b \quad \beta^2 = a\beta + b$$

the two root of a certain quadratic polynomial. These may be complex. Let X, Y satisfy

$$X + Y = x \quad X\alpha + Y\beta = y$$

where again these may be complex. (If $\alpha = \beta$ there may not be such X, Y . You should worry about this case.) Using these constants we see that

$$fr = X\alpha^r + Y\beta^r$$

by a straight forward induction over $r \in \mathbb{N}$.

To extend f into the negative let

$$gr = f(-r)$$

for $r \in \mathbb{N}$ (with the same parameters a, b, x, y). Then a simple manipulation gives

$$bg(r+2) = ag(r+1) - gr$$

and we see that, in general, there is a unique such function g .

There are three of exceptional case that should be considered, and you should do that.

(b) The program

$$I := 0; \textit{while } I < R \textit{ do } Z := A \times X + B \times Y; X := Y; y := Z \textit{ od}$$

can be used to calculate the function for positive r . By preceding this by a conditional (to decide whether r is strictly positive, zero, or strictly negative) gives a program which evaluates the whole function (if the exceptional cases are taken care of). ■

5 The program

$$\frac{[X : \mathbb{R}, Y, Z : \mathbb{N}]}{Y := 1; Z := 0; \textit{while } Y \leq X \textit{ do} \\ Z := Z + 1 \\ Y := Y = 2Y + 1 \\ \textit{od}}$$

will do the job. To see this not that after each pass through the loop we have

$$Y = (Z + 1)^2$$

and the looping stops as soon as $X < Y$, that is when

$$Z^2 \leq X < (Z + 1)^2$$

and then Z will hold $\lfloor \sqrt{x} \rfloor$. ■

For section 3

6 (a) Working in the declaration

$$[X, Y, Z, I : \mathbb{N}]$$

let IT and LP be

$$Z := 0; I := 0 \quad \textit{while } I < X \textit{ do } I := I + 1; Z := Y + Z \textit{ od}$$

respectively. (This is one of the standard ways of converting a head recursion into a while loop.)

(b) Here a state is a 4-tuple (x, y, z, i) and the transition function \mathcal{W} \llbracket LP \rrbracket of the loop is given by

$$\mathcal{W}(x, y, z, i) = \begin{cases} (x, y, z, i) & \text{if } x \leq i \\ \mathcal{W}(x, y, y + z, i + 1) & \text{if } i < x \end{cases}$$

for $x, y, z, i \in \mathbb{N}$.

(c) There are several ways to do this. One method is to first show that

$$\mathcal{W}(x + r, y, z, r) = (x + r, y, xy + z, x + r)$$

holds for all $x, y, z, r \in \mathbb{N}$. This is proved by induction on x . You should do this to see which of the three parameters y, z, r can be held rigid and which must be allowed to vary. Once we have this result we may set $z = 0, r = 0$ to get

$$\mathcal{W}(x, y, 0, 0) = (x, y, xy, x)$$

which is essentially the transition function of the whole program.

(d) If we merely set $r = 0$ in the result above we get

$$\mathcal{W}(x, y, z, 0) = (x, y, xy + z, x)$$

which is the clue. Consider the 3-placed function N specified by

$$N(0, y, z) = z \quad N(x', y, z) = N(x, y, y + z)$$

for $x, y, z \in \mathbb{N}$. This is a tail recursion and easy to program. Since

$$N(x, y, z) = xy + z$$

we get multiplication by setting $z = 0$. (Sometimes an extra variable z used in this way is called an **accumulator**). ■

7 Most of this is dealt with in Solution 2. ■

8 We write $\text{supp}(\sigma)$ for the support of σ , the family of identifiers Z for which

$\sigma Z \neq \perp$. With this we find that

$$\begin{array}{ll}
(a) \quad \delta X = \begin{cases} 1 & \text{if } X \in \text{supp}(\sigma) \\ \perp & \text{if not} \end{cases} & \delta Y = \begin{cases} 0 & \text{if } X, Y \in \text{supp}(\sigma) \\ \perp & \text{if not} \end{cases} \\
(b) \quad \delta X = \begin{cases} 1 & \text{if } X, Y \in \text{supp}(\sigma) \\ \perp & \text{if not} \end{cases} & \delta Y = \begin{cases} 0 & \text{if } Y \in \text{supp}(\sigma) \\ \perp & \text{if not} \end{cases} \\
(c) \quad \delta X = \begin{cases} \sigma Y & \text{if } X \in \text{supp}(\sigma) \\ \perp & \text{if not} \end{cases} & \delta Y = \begin{cases} \sigma Y & \text{if } X \in \text{supp}(\sigma) \\ \perp & \text{if not} \end{cases} \\
(d) \quad \delta X = \begin{cases} 1 & \text{if } X, Y \in \text{supp}(\sigma) \\ \perp & \text{if not} \end{cases} & \delta Y = \sigma Y \\
(e) \quad \delta X = \begin{cases} \sigma X & \text{if } X, Y \in \text{supp}(\sigma) \\ \perp & \text{if not} \end{cases} & \delta Y = \sigma Y \\
(f) \quad \delta X = \begin{cases} \sigma X & \text{if } X \in \text{supp}(\sigma) \\ \perp & \text{if not} \end{cases} & \delta Y = \sigma Y
\end{array}$$

are the required updated states. ■

9 (i) Equivalent.

(ii) Not equivalent. Look at what happens when B returns \perp . The left hand instruction will return \perp (that is fail) but the right hand will execute Q . This can happen even when the left hand instruction is correctly typed, for the state may supply the wrong kind of inputs for B .

(iii – vi) Equivalent.

(ii) Not equivalent. Look at what happens when A returns \perp but B returns true. The left hand program fails but the right hand one executes P . ■

10 (a) With the obvious context the program

$$Y := 0; Z := h(0.P); \text{while } Z \neq 0 \text{ do } Y := Y + 1; Z := h(Y, P) \text{ od}$$

does the job.

(b,c) Consider also the function

$$\mathbb{N}, \text{State} \xrightarrow{F} \text{State}$$

given by

$$F(0, \sigma) = \sigma \quad F(x', \sigma) = \mathcal{Q}(F(x, \sigma))$$

where $\mathcal{Q} = \llbracket Q \rrbracket$. This is a rather simple head recursion in \mathcal{Q} and, if the states σ are not too complicated, it could be described as primitive recursive in \mathcal{Q} .

The function \mathcal{F} of the for loop is given by

$$\mathcal{F}\sigma = F(\llbracket A \rrbracket\sigma, \sigma)$$

which is a substitution instance of F .

(d) We know that bounded search can never escape from the clone \mathcal{K} of Kalmár elementary function which is a rather small part of the clone \mathcal{P} of primitive recursive function. Consider a primitive recursive specification

$$f(0, p) = gp \quad f(x', p) = h(x, p, f(x, p))$$

where the data functions g, h are primitive recursive. The for loop

```

 $I := 0; F := g(P);$ 
For  $X$  do
   $F := h(I, P, F); I := I + 1$ 
od

```

computes f . Thus a for loop can escape from \mathcal{K} (but can not escape from \mathcal{P}).

[Try to sort out a more informal explanation in terms of space resource] ■

For section 4

11 We proceed by a base/step induction on x with variation of i and and with p, t held rigid.

The base case, $x = 0$, is immediate.

For the induction step, $x \mapsto x'$ let

$$t^+ = H(i, p, t)$$

so that

$$\mathcal{H}(i, x' + i, p, t^+) = \mathcal{H}(i', x' + i, p, h(i, p, t^+)) = \mathcal{H}(i', x + i', p, H(i', p, t))$$

using first the specification of \mathcal{H} and then that of H . But now we can invoke the induction hypothesis to get

$$\mathcal{H}(i, x' + i, p, t^+) = \mathcal{H}(i', x + i', p, H(i', p, t)) = (x + i', x + i', p, H(x + i', p, t))$$

and hence

$$\mathcal{H}(i, x' + i, p, H(i, p, t)) = \mathcal{H}(i, x' + i, p, t^+) = (x' + i, x' + i, p, H(x' + i, p, t))$$

as required. ■

12 This is done in Solution 6. ■

13 (a) We find that

$$\mathbb{P}, \mathbb{T} \xrightarrow{h} \mathbb{T} \qquad \mathbb{N}, \mathbb{P}, \mathbb{T} \xrightarrow{H, T} \mathbb{T}$$

are the relevant types.

(b) We proceed by induction on x with variation of y and with p and t rigid. The base case, $x = 0$, is immediate. For the induction step, $x \mapsto x'$ we have

$$\begin{aligned} T(x', p, H(y, p, t)) &= T(x, p, h(p, H(y, p, t))) \\ &= T(x, p, H(y', p, t)) \\ &= H(x + y', p, t) \qquad = H(x' + y, p, t) \end{aligned}$$

using the specification of T , the specification of H , the induction hypothesis, and a trivial manipulation, in that order.

(c) This follows by setting $y = 0$ in the equality of part (b). (Notice that doesn't mean we could ask for something simpler in (b).)

(d) Given the parameter p we have $hp \in \mathbb{T}'$ and this function can be iterated. We find that

$$H(x, p, t) = (hp)^x t = T(x, pt)$$

and the two functions H, T are merely an outer and an inner iteration. ■

14 (a) We find that

$$\mathbb{N}, \mathbb{P}, \mathbb{T} \xrightarrow{h, H, T} \mathbb{T}$$

for all three functions.

(b) For the left hand identity we proceed by induction on x with the parameters held rigid.

The base case, $x = 0$, is straight forward. For the induction step, $x \mapsto x'$ we have

$$\begin{aligned} H(x', p, h(i, p, t)) &= h(x, p, H(x, p, h(i, p, t))) \\ &= h(x, p, h(i, p, H(x, p, t))) \\ &= h(i, p, h(x, p, H(x, p, t))) = h(i, p, H(x', p, t)) \end{aligned}$$

using the specification of H , the induction hypothesis, the given commuting property of h , and a second use of the specification of H .

The right hand identity follows by a similar proof.

(c) For fixed p, t we show

$$H(x, p, t) = T(x, p, t)$$

by induction on x .

The base case, $x = 0$, is immediate. For the induction step, $x \mapsto x'$ we have

$$\begin{aligned} H(x', p, t) &= h(x, p, H(x, p, t)) \\ &= h(x, p, T(x, p, t)) \\ &= T(x, p, h(x, p, t)) = T(x', p, t) \end{aligned}$$

using the specification of H , the induction hypothesis, the right hand result of part (b) (for the case $x = i$), and the specification of T . This induction step can be reorganized to use the left hand result of part (b).

(d) For $x \in \mathbb{N}$ and $p \in \mathbb{P}$ let

$$h_{(x,p)} = h(x, p, \cdot)$$

to obtain a function in $Tblock'$ which can be iterated. We find that

$$\begin{aligned} H(x, p, \cdot) &= h_{(x-1,p)} \circ h_{(x-2,p)} \circ \cdots \circ h_{(1,p)} \circ h_{(0,p)} \\ T(x, p, \cdot) &= h_{(0,p)} \circ h_{(1,p)} \circ \cdots \circ h_{(x-2,p)} \circ h_{(x-1,p)} \end{aligned}$$

with an obvious modification for $x = 0$. The given commuting property of h means we can shuffle round the order of the components $h_{(i,p)}$, and so get the equality of $H(x, p, \cdot)$ and $T(x, p, \cdot)$. ■