4. Proof

Simon J. Thompson[1]

4.1 Introduction

In this chapter we examine ways in which functional programs can be proved correct. For a number of reasons this is easier for functional than for imperative programs. In the simplest cases functional programs are equations, so the language documents itself, as it were. Beyond this we often have a higher-level expression of properties, by means of equations between functions rather than values. We can also express properties which cannot simply be discussed for imperative programs, using notations for lists and other algebraic data types, for instance.

Apart from the general observation that proofs carry over directly to implicitly parallel functional systems, what is the particular relevance of proof to parallel functional programming? Two particular points are emphasised in this chapter.

- Lazy evaluation gives infinite and partial data structures, which can be viewed as describing the data flowing around deterministic networks of processes (see Section 3.5). Section 4.8.3 gives a proof that a process network produces the list of factorials; the bisimulation method used here forms a link with verification techniques for process algebras, which give a declarative treatment of parallelism and non-determinism and which are surveyed in Section 4.8.4.
- There is an important thread in the development of functional programming, going back at least to Backus' FP [27], which argues that it is best to eschew explicit recursion and to program using a fixed set of higher-order, polymorphic combinators, which are related to programming skeletons (see Section 3.5). The properties of these combinators can be seen as logical laws in an algebra of programming [51]. These laws can also be seen to have intensional content, with a transformation from left- to right-hand side representing a change in the cost of an operation, or in the parallelism implicit there (see Chapter 8 for more details of cost modelling). This topic is examined further in Section 4.9.3.

The equational model, explored in Section 4.2, gives the spirit of functional program verification, but it needs to be modified and strengthened in various ways in order to apply to a full functional language. The pattern of the chapter will be to give a succession of refinements of the logic as further features are added to the language. These we look at now.

[1] Computing Laboratory, University of Kent at Canterbury, UK.

The defining forms of languages are more complex than simple equations. Section 4.3 looks at conditional definitions (using "guards"), pattern matching and local definitions (in `let` and `where` clauses) each of which adds complications, not least when the features interact.

Reasoning cannot be completely equational. We need to be able to reason by cases, and in general to be able to prove properties of functions defined by recursion. Structural induction is the mechanism: it is discussed in Section 4.4, and illustrated by a correctness proof for a miniature compiler in Section 4.5.

With general recursion, examined in Section 4.6 and which is a feature of all languages in the current mainstream, comes the possibility of non-termination of evaluation. In a non-strict language general recursion has the more profound effect of introducing infinite and partial lists and other data structures.

In both strict and non-strict languages this possibility means in turn that in interpreting the meaning of programs we are forced to introduce extra values at each type. This plainly affects the way in which the logic is expressed, and its relation to familiar properties of, say, the integers. The background to this is explored in Section 4.7.

Section 4.8 gives an overview of the two principal approaches to giving meanings to programs – denotational semantics and operational semantics – and their implications for program verification. A proof of correctness for a function defined by general recursion exemplifies the denotational approach, whilst operational semantics underpins a proof that a lazy process network generates the list of factorials. This last proof forms a link with the process algebra approach to parallelism, which is also discussed in this section.

Because of the complications which non-termination brings, there has been recent interest in terminating languages and these are introduced in Section 4.9. Of particular relevance here is the transformational approach of Backus, Bird and others.

A fully-fledged language will allow users to interact with the environment in various ways, but at its simplest by reading input and writing output. This is supported in a variety of ways, including the side-effecting functions of Standard ML (SML) and the monads of Haskell 98. SML also allows mutable references and exceptions. In this chapter we cover only the pure parts of languages, but refer readers to [221] for a perspicacious discussion of program verification for various forms of input/output including monadic I/O. Recent work on modelling SML-style references can be found in [465].

This chapter does not try to address all the work on verification of parallel imperative programs: Sections 8.9 and 8.10 of the exhaustive survey [140] more than do justice to this topic, and put it in the context of imperative program verification in general. On the other hand, links with process algebra are examined in Section 4.8.4.

4.2 The Basis of Functional Programming: Equations

In this section we examine the basis of functional programming and show how the definitions of a simple functional program can be interpreted as logical equations. An examination of how this approach can be modified and extended to work in general forms the main part of the chapter.

A functional program consists of a collection of definitions of functions and other values. An example program using the notation of the Haskell language [448] is

```
test :: Integer
test = 42

id :: a -> a
id x = x

plusOne :: Integer -> Integer
plusOne n = (n+1)

minusOne :: Integer -> Integer
minusOne n = (n-1)
```

Execution of a program consists of evaluating an expression which uses the functions and other objects defined in the program (together with the built in operations of the language). Evaluation works by the replacement of sub-expressions by their values, and is complete when a value is produced. For instance, evaluation of

```
plusOne (minusOne test)
```

will proceed thus:

```
plusOne (minusOne test)
 => (minusOne test) + 1
 => (test - 1) + 1
 => (42 - 1) + 1
 => 41 + 1
 => 42
```

where it can be seen that at each stage of the evaluation one of the defining equations is used to rewrite a sub-expression which matches the left-hand side of a definition, like

```
minusOne test
```

to the corresponding right-hand side,

```
test - 1
```

The model of evaluation for a real language such as Haskell or ML is somewhat more complex; this will be reflected by the discussion in subsequent sections.

In each step of an evaluation such as this equals are replaced by equals, and this points to the basis of a logical approach to reading functional programs. The use of the equals sign in function definitions is indeed suggestive, and we can read the definitions as **logical statements** of the properties of the defined functions, thus:

```
id x ≡ x                                                    (id.1)
```

for all `x` of type `t`, and so on. Note that we have used the symbol "≡" here for logical equality to distinguish it from both the "definitional" equality used to define objects in the language, `=`, and the "calculational" Boolean equality operation of the language, `==`.

Logical equations like these can be manipulated using the rules of logic in the standard way, so that we can deduce, for instance, that

```
id (id y)
  ≡ {by substituting id y for x in (id.1)}
id y
  ≡ {by substituting y for x in (id.1)}
y
```

In linear proofs we shall use the format above, in which the justification for each equality step of the proof is included in braces $\{\cdots\}$.

So, we see a model for verification of functional programs which uses the defining equations as logical equations, and the logical laws for equality: reflexivity, symmetry, transitivity and substitution:

$$\frac{\texttt{P(a)} \quad \texttt{a} \equiv \texttt{b}}{\texttt{P(b)}}\,(\mathit{Subst})$$

to make deductions. Note that in giving the substitution rule we have used the convention that `P(a)` means an expression `P` in which `a` occurs; the appearance of `P(b)` below the line means that the occurrences of `a` in `P` in have been replaced by `b`. An alternative notation which we use later in the chapter is `P[b/a]` which is used to denote "`b` substituted for `a` in `P`".

The logical versions of the definitions given here contain free variables, namely the variables of the definitions. In the remainder of the chapter we will also use a closed form given by taking the universal quantification over these variables. `(id.1)` will then take the form `(∀x::a)(id x ≡ x)` for example.

4.3 Pattern Matching, Cases and Local Definitions

The purely equational definition style of Section 4.2 can be made to accommodate case switches, local definitions and pattern matching by means of the appropriate higher-order combinators. Indeed, this is one way of interpreting the work of Bird and others, discussed further in Section 4.9.3. However, for reasons of readability and conciseness, most languages offer syntactic support for these facilities, and with this additional syntax comes the task of giving it a logical explanation.

This section gives an overview of how pattern matching, cases and local definitions are rendered logically; a more detailed examination can be found in [551], which addresses the question for Miranda. Note that here we are still considering a small (terminating) language, rather than a full language.

4.3.1 Pattern Matching

Pattern matching serves to distinguish cases, as in

```
isEmptyList :: [a] -> [a]
isEmptyList [] = True
isEmptyList _  = False
```

(where "`_`" is a wildcard pattern, matching anything), and also to allow access to the components of a compound object

```
tail :: [a] -> [a]
tail []     = []
tail (x:xs) = xs
```

In the example of `tail`, where the patterns do not overlap (are exclusive) and cover all eventualities (are exhaustive), the definitions can be read as logical equations.

In the general case, we need to take account of the **sequential** interpretation which is usually applied to them. Looking at `isEmptyList`, the second equation in which the "`_`" will match any value will only be applied should the first clause not apply. We therefore need to give a description of the complement of a pattern, here `[]`, over which the remaining equations hold. The complement of `[]` will be the non-empty list, `(x:xs)`, and so we can rewrite the definition of the function to give its logical form thus:

```
isEmptyList []     ≡ True
isEmptyList (x:xs) ≡ False
```

As another example, consider the pattern `(x:y:ys)`. This will match lists with two or more elements, and its complement is given by the two patterns `[]` and `[_]`. The full details of the way in which pattern matching definitions can be translated are to be found in [551].

4.3.2 Cases

Definitions can have alternatives depending on the (Boolean) values of guards, in a Haskell style,

```
f args
    | g₁        = e₁
    | g₂        = e₂
      ...
    | otherwise = e
```

If the (actual values of the) parameters `args` satisfy g_1 then the result of `f args` is e_1; should g_1 be `False` then if g_2 is `True`, e_2 is the result, and so on. In logical form we then have

$(g_1 \equiv \texttt{True} \Rightarrow \texttt{f args} \equiv e_1) \wedge$
$((g_1 \equiv \texttt{False} \wedge g_2 \equiv \texttt{True}) \Rightarrow \texttt{f args} \equiv e_2) \wedge \ldots$

which renders the definition as the conjunction of a set of conditional equations.

4.3.3 Local Definitions

A local definition, introduced either by a `let` or a `where` introduces a name whose scope is restricted. An example is given by the schematic

```
f :: a₁ -> a₂
f x = e
      where
      g :: a₃ -> a₄
      g y = e'
```

The function `g` is in scope in the expression `e` as well as the `where` clause. It is also important to realise that its definition will, in general, depend upon the parameter `x`. It is translated thus

$(\forall x :: a_1)(\exists g :: a_3 \texttt{ -> } a_4)((\forall y :: a_3)(g\ y \equiv e') \wedge f\ x \equiv e)$

in which the locally defined value(s) are existentially quantified, and the universal quantification over the argument values for `f` and `g` are shown explicitly.

4.3.4 Further Issues

The features discussed in this section can, when they appear in real programming languages such as Haskell, have complex interactions. For instance, it is not necessary to have an `otherwise` case in a guarded equation, so that it is possible for none of the guards to hold for a particular set of arguments. In this situation, the next guarded equation (and therefore pattern match) has to be examined, and this is particularly difficult to explain when the guards also refer to local definitions – an example is presented in [551].

The translation given here goes beyond the equational, giving axioms which involve arbitrarily deep alternations of quantifiers. In practice these quantifiers will be stripped off, allowing conditional equational reasoning take place; the effect of the quantifications is to ensure that the scoping rules of the language are obeyed, while the conditions reflect the guards in the definitions of the language. Pattern matching is supported by the substitution mechanism of the logic.

4.4 Structural Induction and Recursion

In this section we consider how to strengthen our language to accommodate recursively defined functions and types while retaining the property that all computations will terminate.

At the heart of modern functional programming languages are built-in types of lists and a facility to define "algebraic" data types built by the application of constructors. If we wish to build a simple-minded representation of integer arithmetic expressions — as part of a calculator or a compiler, say — we might write, using Haskell notation

```
data IntExp = Literal Int |
              Binary Op IntExp IntExp

data Op = Add | Sub | Mul
```

which describes a type whose members take two forms, built by the two constructors of the type, `Literal` and `Binary`.

- The first is `Literal n`, where `n` is an `Int` (integer).
- The second form is `Binary op ex1 ex2` where `ex1` and `ex2` are themselves `IntExp`s and `op` is one of `Add`, `Sub` or `Mul` (representing three binary arithmetic operators).

An example of the type, representing the arithmetic expression (4+3)-5, is

```
Binary Sub (Binary Add (Literal 4) (Literal 3)) (Literal 5)
```

To define a function over `Op` it is sufficient to give its value at the three possible inputs, so that

```
opValue :: Op -> (Int -> Int -> Int)

opValue Add = (+)
opValue Sub = (-)
opValue Mul = (*)
```

serves to interpret the arithmetic operators. In a similar way, if we wish to prove that some logical property holds for all operators it is sufficient to prove that the property holds for the three values of the type.

Now, the type `IntExp` is rather more complicated, since it is recursively defined, and has an infinite number of members. However, we know that the only ways that elements are constructed are by means of a finite number of applications of the constructors of the type. This means that an arbitrary element of the type will take one of the forms

```
Literal n
Binary op ex1 ex2
```

where `ex1` and `ex2` are themselves elements of `IntExp`.

Because every element is built up in this way, we can deduce how to define functions over `IntExp` and to prove that properties hold of all elements of `IntExp`. To define a function we use **structural recursion**, as exemplified by a function to evaluate an arithmetic expression:

```
eval :: IntExp -> Int

eval (Literal int) = int                                (eval.1)
eval (Binary op ex1 ex2)
  = opValue op (eval ex1) (eval ex2)                    (eval.2)
```

Here we see the pattern of definition in which we

- give the result at `Literal int` outright; and
- give the result at `Binary op ex1 ex2` using the results already defined for `ex1` and `ex2` (as well as other components of the data value, here `op`).

It can be seen that a finite number of recursive calls will result in calls to the `Literal` case, so that functions defined in this way will be total.

In an analogous way, we can use structural induction to prove a property for all `IntExp`s. This principle is stated now.

Structural induction. To prove `P(e)` for all `e` in `IntExp` we need to show that

- **Base case.** The property `P(Literal int)` holds for all `int`.
- **Induction case.** The property `P(Binary op ex1 ex2)` holds *on the assumption that* `P(ex1)` and `P(ex2)` hold.

Given any `IntExp t` we can see that a finite number of applications of the induction case will lead us back to the base case, and thus establish that `P(t)` holds.

In the next section we give examples of various functions defined by structural recursion together with verification using structural induction over the `IntExp` type.

4.5 Case Study: a Compiler Correctness Proof

In this section we give a proof of correctness of a tiny compiler for arithmetic expressions using structural induction over the type of expressions, given by the algebraic data type `IntExp`. In developing the proof we explore some of the pragmatics of finding proofs.

It is instructive to compare this program and proof developed in a functional context with a similar problem programmed in a modern imperative language such as C++, Java or Modula 3. The advantage of the approach here is that modern functional languages contain explicit representations of recursive data types, and so a proof of a program property can refer explicitly to the forms of data values. In contrast, a stack in an imperative language will either be represented by a dynamic data structure, built using pointers, or by an array, with the attendant problems of working with a concrete representation of a stack rather than an appropriately abstract view. In either case it is not so easy to see how a proof could be written, indeed the most appropriate model might be to develop the imperative program by refinement from the verified functional program presented here.

4.5.1 The Compiler and Stack Machine

The program is given in Figure 4.1, in two halves. In the first half we reiterate the definitions of the `IntExp` type and its evaluation function `eval`, which is defined by structural recursion over `IntExp`.

In the second half of the figure we give a model of a stack machine which is used to evaluate the expressions. The machine operates over a stack of integers, hence the definition

```
type Stack = [Int]
```

The instructions for the machine are given by the type `Code`, which has two operations, namely to push an element (`PushLit`) onto the stack and to perform an evaluation of an operation (`DoBinary`) using the top elements of the stack as arguments.

An expression is converted into a `Program`, that is a list of `Code`, by `compile`. The `compile` function compiles a literal in the obvious way, and for an operator expression, the compiled code consists of the compiled code for the two expressions, concatenated by the list operator `++`, with the appropriate binary operator invocation appended.

The operation of the machine itself is described by

```
run :: Program -> Stack -> Stack
```

and from that definition it can be seen that if the stack fails to have at least two elements on operator evaluation, execution will be halted and the stack cleared.

4.5.2 Formulating the Goal

The intended effect of the compiler is to produce code (for `e`) which when run puts the value of `e` on the stack. In formal terms,

```
run (compile e) [] ≡ [eval e]                         (compGoal.1)
```

Now, we could look for a proof of this by structural induction over `e`, but this will fail. We can explain this failure from two different points of view.

Looking first at the problem itself, we can see that in fact the compiler and machine have a rather more general property: no matter what the initial configuration of the stack, the result of the run should be to place the value of the expression on the top of the stack:

```
run (compile e) stack ≡ (eval e : stack)
```

This is still not general enough, since it talks about complete computations – what if the code is followed by more program? The effect should be to evaluate `e` and place its result on the stack prior to executing the remaining program. We thus reach the final formulation of the goal

```
run (compile e ++ program) stack
    ≡ run program (eval e : stack)                    (compGoal.2)
```

An alternative view of the difficulty comes from looking at the failed proof attempt: the induction hypothesis turns out not to be powerful enough to give what is required. This happens in the case of a binary operation, when we try to prove `(compGoal.1)` where `e` is, for example, `Binary Add ex1 ex2`. In this case we need to prove that

```
run (compile ex1 ++ compile ex2 ++ [DoBinary Add]) ≡ [eval e]
```

so that we will need a hypothesis about `compile ex1` *in context*

```
run (compile ex1 ++ ...)
```

rather than in isolation.

This leads us to formulate the generalisation `(compGoal.2)` — this is examined in the next section and again it is shown there how a failed proof attempt leads to an suitable formalisation of the induction hypothesis.

A guide to the form of hypothesis is often given by the form taken by the definitions of the functions under scrutiny; we will discuss this point after giving the full proof in next section.

4.5.3 The Proof

Our goal is to prove `(compGoal.2)` for all values of `e`, `program` and `stack`. As a first attempt we might try to prove `(compGoal.2)` by induction over `e`, for arbitrary `program` and `stack`, but again this will fail. This happens because the induction hypothesis will be used at different values of `stack` and `program`, so that the goal for the inductive proof is to show by structural induction on `e` that

```
(∀program,stack)(run (compile e ++ program) stack
    ≡ run program (eval e : stack))                         (goal)
```

holds for all `e`.

The proof is given in Figure 4.2 and follows the principle of structural induction for `IntExp` presented in Section 4.4 above. In the first part we prove the base case:

```
run (compile (Literal int) ++ program) stack
    ≡ run program (eval (Literal int) : stack)          (base)
```

for arbitrary `program,stack`, thus giving the base case of `(goal)`. The proof proceeds by separately rewriting the left- and right-hand sides of `(base)` to the same value.

In the second part we show

```
run (compile (Binary op ex1 ex2) ++ program) stack
    ≡ run program (eval (Binary op ex1 ex2) : stack)      (ind)
```

for arbitrary `program,stack` using the induction hypotheses for `ex1`:

```
(∀program,stack)(run (compile ex1 ++ program) stack
    ≡ run program (eval ex1 : stack))                     (hyp)
```

and `ex2`. It is instructive to observe that in the proof the induction hypothesis for `ex1`, `(hyp)`, is used with the expression

```
compile ex2 ++ [DoBinary op] ++ program
```

substituted for `program`, and that for `ex2` is used in a similar way. Again the proof proceeds by separately rewriting the left- and right-hand sides of `(ind)`.

The third part of Figure 4.2 shows how our original goal, `(compGoal.1)` is a consequence of the more general result `(compGoal.2)`.

How might we be led to the goal `(goal)` by the form of the program itself? If we examine the definition of `run` we can see that in the recursive calls `(run.2)` and `(run.3)` the `stack` parameter is modified. This indicates that the `stack` cannot be expected to be a parameter of the proof, and so that the general formulation of the induction hypothesis will have to include all possible values of the stack parameter.

4.6 General Recursion

In the preceding sections we saw how structural recursion and induction can be used to define and verify programs over algebraic data types. Functions defined in this way are manifestly total, but there remains the question of whether these limited forms of recursion and induction are adequate in practice. An example going beyond structural recursion over `IntExp` is a function to re-arrange arithmetic expressions so that the additions which they contain are associated to the left, transforming

`(4+2)+(3+(7+9))` to `(((4+2)+3)+7)+9`

The function is defined thus:

```
lAssoc :: IntExp -> IntExp

lAssoc (Literal n) = Literal n
lAssoc (Binary Sub ex1 ex2)
    = Binary Sub (lAssoc ex1) (lAssoc ex2)
lAssoc (Binary Add ex1 (Binary Add ex3 ex4))
    = lAssoc (Binary Add (Binary Add ex1 ex3) ex4)   (lAssoc.1)
lAssoc (Binary Add ex1 ex2)
    = Binary Add (lAssoc ex1) (lAssoc ex2)
```

(where the `Mul` case has been omitted). Each clause is structurally recursive, except for `(lAssoc.1)`, in which the top-level expression `ex1+(ex3+ex4)` is transformed to `(ex1+ex3)+ex4`. Once this transformation has been effected, it is necessary to re-examine the whole re-arranged expression, and not just the components of the original. The reader might like to experiment with the example expression to convince herself of the necessity of making a definition of this form, rather than a structural recursion.

Now, what is the lesson of examples like this for the design of functional programming languages and for verification of systems written in them? There are broadly two schools of thought.

The predominant view is to accept that a language should allow arbitrary recursion in the definitions of functions (and perhaps other objects). Mainstream languages such as Haskell, Miranda and Standard ML are all of this kind. With arbitrary recursion come a number of consequences.

- The semantics of the language becomes more complex, since it must now contain an account of the possible non-termination of programs.
- Moreover, the evaluation mechanism becomes significant. If all programs terminate, then the order in which programs are evaluated is not an issue; if non-termination is possible then strict and lazy evaluation strategies differ, and thus give strict and non-strict languages different semantics.
- As far as the topic of this chapter is concerned, the complexity of the semantics is reflected in the logic needed to reason about the language, for both strict and non-strict languages.

For these reasons there has been recent interest in terminating languages — Turner's notion of "strong" functional languages [564] — because such languages both have a simpler proof theory and have full freedom of choice for evaluation strategy, which is of course of relevance to the field of parallel functional programming.

In the remainder of this chapter we will explore the effect of these two alternatives for functional program verification, first looking at the mainstream, partial, languages.

4.7 Partial Languages

This section gives an informal overview of the effect of admitting general recursion into a programming language, and emphasises the consequent split between strict and non-strict languages. This serves as an introduction to the overview of the semantic basis of languages with partiality in the section to come.

4.7.1 Strict Languages

In a strict language such as (the pure subset of) Standard ML arbitrary forms of recursive definitions are allowed for functions. A definition of the form

```
undefFun :: a -> a
undefFun x = undefFun x                          (undefFun.1)
```

(using Haskell-style syntax) has the effect of forcing there to be an undefined element at every type. What effect does this have for evaluation and for the logic? Take the example function

```
const :: a -> b -> a
const x y = x
```

and consider its logical translation. Our earlier work suggests that we translate it by

`const x y` $\equiv$ `x` (const.1)

but we need to be careful what is substituted for the variables `x` and `y`. If we take `x` to be `3` and `y` to be `undefFun 4` then it appears that

`const 3 (undefFun 4)` $\equiv$ `3`

This is contrary to the rule for evaluation which states that arguments need to be evaluated prior being passed to functions, and which means that `(undefFun.1)` should be undefined when applied to `undefFun 4`. The translation `(const.1)` can therefore only apply to **values** (of type `Int`) rather than arbitrary **expressions** of that type as was the case earlier. This can be made clear by re-expressing `(const.1)` thus:

$(\forall_v$ `x,y)(const x y` $\equiv$ `x)`

where the subscript in the quantifier "$\forall_v$" serves as a reminder that the quantifier ranges over all (defined) values rather than all expressions including those which denote an undefined computation.

4.7.2 Non-Strict languages

In a non-strict language like Haskell the definition of `undefFun` in `(undefFun.1)` also gives rise to an undefined element at each type. This does not however affect the translation of `const` given in `(const.1)` above, since in a non-strict language expressions are passed *unevaluated* to functions. In other words, the evaluation mechanism can truly be seen to be one of substitution of expressions for expressions. (For efficiency, this "call by name" strategy will be implemented by a "call by need" discipline under which the results of computations are shared.)

Nevertheless, the presence of an undefined expression in each type has its effect. We accept as a law the assertion that for all integers `x`

```
x+1 > x
```

but this will not be the case if `x` is an undefined computation. We will therefore have to make the distinction between defined values and all expressions as in Section 4.7.1.

The result of combining lazy evaluation and general recursion are more profound than for a strict language, since data structures can become partial or infinite. The effect of

```
nums = from 1
from n = n : from (n+1)
```

is to define the infinite list of positive integers, `[1,2,3,...]`. If `nums` is passed to a function, then it is substituted unevaluated, and parts of it are evaluated when and if they are required:

```
sft :: [Int] -> Int
stf (x:y:_) = x+y                                        (sft.1)

sft nums
  => sft (from 1)
  => sft (1 : from 2)
  => sft (1 : 2 : from 3)
```

At this point the pattern match in `(sft.1)` can be performed, giving the result `3`. Our interpretation therefore needs to include such infinite lists, as well as "partial" lists such as `(2:undefFun 2)`. Note that under a strict interpretation all infinite and partial lists are identified with the undefined list, since they all lead to non-terminating computations.

In order to give a proper account of the behaviour of languages with non-termination we now look at the ways in which a formal or mathematical semantics can be given to a programming language.

4.8 Semantic Approaches

This section surveys the two semantic approaches to functional programming languages with the aim of motivating the logical rules to which the semantics lead.

4.8.1 Denotational Semantics

Under a denotational semantics, as introduced in the textbook [586], the objects of a programming language — both terminating and non-terminating — are modelled by the elements of a **domain**. A domain is a partially ordered structure, where the partial order reflects the degree of definedness of the elements, with the totally undefined object, $\bot$, below everything: $\bot \sqsubseteq \mathtt{x}$. Recursion, as in the definition

$$\mathtt{f = C[f]}$$

can then be explained by first looking at the sequence of approximations, $\mathtt{f_n}$, with

$$\mathtt{f_0} \equiv \bot$$

and

$$\mathtt{f_{n+1} = C[f_n]}$$

A domain also carries a notion of limit for sequences (or indeed more general "directed sets"), so that the meaning of `f`, $[\![\mathtt{f}]\!]$, is taken to be the limit of this sequence of approximations:

$$\mathtt{f} \equiv \bigsqcup_{\mathtt{n}} \mathtt{f_n}$$

Another way of seeing this is that $[\![\mathtt{f}]\!]$ is the **least fixed point** of the operation

$$\lambda\mathtt{f.C[f]}$$

with a domain having sufficient structure to provide fixed points of (monotone) operators over them.

All the data types of a functional language can be modelled in such a way, and reasoning over domains is characterised by fixed-point induction, which captures the fact that a recursively defined function is the limit of a sequence. Before stating the principle, an auxiliary definition is needed.

A predicate `P` is called **inclusive** if it is closed under taking limits, broadly speaking. Winskel, [586], provides a more detailed characterisation of this, together with sufficient conditions for a formula to be an inclusive predicate.

> **Fixed-point induction.** If `P` is an inclusive predicate and if `f` is defined as above, then if
> - $\mathtt{P(\bot)}$ holds, and (FPI.1)
> - $\mathtt{P(f_n)}$ implies $\mathtt{P(f_{n+1})}$; (FPI.2)
>
> then `P` holds of the limit of the sequence, that is `P(f)`.

As an example we look again at the `lAssoc` function, defined in Section 4.6 above. We would like to show that rearranging an expression will not change its value, that is

$$(\forall \mathtt{e})(\mathtt{eval\ (lAssoc\ e)} \equiv \mathtt{eval\ e}) \qquad \mathtt{P_0(lAssoc)}$$

(where `eval` is defined in Section 4.4). Equations are inclusive, but unfortunately we cannot prove the inductive goals in this case. Take the case of `(FPI.1)`; this states that the property should hold when the function `lAssoc` is replaced by the totally undefined function, $\perp$, and so that we should prove

```
(∀e)(eval (⊥ e) ≡ eval e)
```

which, since the $\perp$ function is undefined on every argument, is equivalent to

```
(∀e)(eval ⊥ ≡ eval e)
```

which is plainly not the case.

We can modify the property to say that *if the result is defined* then the equality holds, namely,

```
(∀e)((lAssoc e ≡ ⊥) \/ eval (lAssoc e) ≡ eval e)     P(lAssoc)
```

It is interesting to see that this is a **partial correctness** property, predicated on the termination of the `lAssoc` function, for which we have to prove a separate termination result. We discuss termination presently.

To establish this result we have to prove `(FPI.1)` and `(FPI.2)` for this property. A proof of `(FPI.1)` is straightforward, since `P(⊥)` states:

```
(∀e)((⊥ e ≡ ⊥) \/ eval (⊥ e) ≡ eval e)
```

and `⊥ e ≡ ⊥` holds, as discussed earlier. A proof of `(FPI.2)` requires that we show that `P(lAssoc`$_n$`)` implies `P(lAssoc`$_{n+1}$`)` where (omitting the `Mul` case),

```
lAssoc_{n+1} (Literal n) = Literal n                        (lA.1)
lAssoc_{n+1} (Binary Sub ex1 ex2)
    = Binary Sub (lAssoc_n ex1) (lAssoc_n ex2)              (lA.2)
lAssoc_{n+1} (Binary Add ex1 (Binary Add ex3 ex4))
    = lAssoc_n (Binary Add (Binary Add ex1 ex3) ex4)        (lA.3)
lAssoc_{n+1} (Binary Add ex1 ex2)
    = Binary Add (lAssoc_n ex1) (lAssoc_n ex2)              (lA.4)
```

Now, our goal is to prove that

```
(∀e)((lAssoc_{n+1} e ≡ ⊥) \/ eval (lAssoc_{n+1} e) ≡ eval e)
```

on the assumption that

```
(∀e)((lAssoc_n e ≡ ⊥) \/ eval (lAssoc_n e) ≡ eval e)
```

We look at the cases of the definition in turn. For a literal we have by `(lA.1)`

```
lAssoc_{n+1} (Literal n) ≡ Literal n
```

from which we conclude immediately that

```
eval (lAssoc_{n+1} (Literal n)) ≡ eval (Literal n)
```

Now, looking at subtraction, and assuming that the function terminates, we have

`eval (lAssoc`$_{n+1}$` (Binary Sub ex1 ex2))`
    $\equiv$ { by `(lA.2)` }
`eval (Binary Sub (lAssoc`$_n$` ex1) (lAssoc`$_n$` ex2))`
    $\equiv$ { by definition of `eval` }
`eval (lAssoc`$_n$` ex1) - eval (lAssoc`$_n$` ex2)`
    $\equiv$ { by termination and the induction hypothesis}
`eval ex1 - eval ex2`
    $\equiv$ { by definition of `eval` }
`eval (Binary Sub ex1 ex2)`

The tricky case is `(lA.3)`, which is the non-structurally recursive clause. Now, again assuming termination, we have

`eval (lAssoc`$_{n+1}$` (Binary Add ex1 (Binary Add ex3 ex4)))`
    $\equiv$ { by `(lA.3)` }
`eval (lAssoc`$_n$` (Binary Add (Binary Add ex1 ex3) ex4))`
    $\equiv$ { by termination and the induction hypothesis}
`eval ((Binary Add (Binary Add ex1 ex3) ex4))`
    $\equiv$ { by the associativity of `+` }
`eval (Binary Add ex1 (Binary Add ex3 ex4))`

The final case – which corresponds to `(lA.4)` – follows exactly the proof for the `(lA.2)` case, with `Add` replacing `Sub`. This establishes the induction step, and so the result itself.

How do we prove that `lAssoc` terminates on all arguments? We need to have some "measure of progress" in the recursive calls. In all calls but `(lAssoc.1)` the recursive calls are on structurally smaller expressions, but in `(lAssoc.1)` the call is to an expression containing the same number of operators. What is changed in the recursive call is the arrangement of the expression, and it is easy to see that on the right hand side of the `Add` in the recursive call there are fewer applications of `Add` than in the same position on the left hand side:

```
     +                         +
    / \                       / \
  e1   +                     +   e3
      / \                   / \
     e2 e3                 e1 e2
```

This reduction means that there can only be a finite number of repeated calls to `(lAssoc.1)` before one of the structural cases is used. Informally, what we have done is to give an ordering over the expressions which is **well-founded**, that is has no infinite descending chains (like the chain $-1 > -2 > \ldots > -n > \ldots$ over the integers). A recursion will terminate precisely when it can be shown to follow a well-founded ordering.

Further details about denotational semantics can be found in [586, 442]. We also refer back to denotational semantics at the end of Section 4.8.3

4.8.2 Operational Semantics

The structured ("SOS") style of operational semantics pioneered by Plotkin describes a programming language by means of deduction rules which explain how expressions are evaluated. This style has been used to describe real languages, notably Standard ML [396], and arguably it gives a more readable and concise description of a language than a denotational semantics. The account given in this section relies on Gordon's thesis, [221], which serves as an introduction to the way that these ideas are applied to the description of functional programming languages.

SOS descriptions give reduction rules (describing "one step" of the computation), as in

$$((\lambda x.M)N) \longrightarrow M[N/x]$$

or can provide a description of the evaluation of an expression to a value (the "big step" rules), thus:

$$\frac{L \Longrightarrow (\lambda x.M) \qquad M[N/x] \Longrightarrow V}{(L\ N) \Longrightarrow V}$$

These rules are related, with $\Longrightarrow$ representing arbitrarily many steps under the relation $\longrightarrow$. From these rules an equality relation can be generated: two expressions are equal, $L \simeq M$, if whatever context $C[\_]$ they are placed in, $C[L] \Longrightarrow V$ if and only if $C[M] \Longrightarrow V$. Now, the issue becomes one of finding ways of deducing, for given expressions L and M, that $L \simeq M$ holds. Abramsky [5] had the insight that this relation resembled the bisimulations of process calculi. This characterises the equivalence as a *greatest* fixed point.

Rather than look at the general theory of bisimulations, we will look here at how it applies to infinite lists. We take an infinite example because in the finite case the flavour of proof is similar to the denotational style, so that the proof of correctness for `lAssoc` would follow similar lines to that in Section 4.8.1; it is in the infinite case that a distinctive style emerges.

The equality relation over infinite lists, "$\simeq$", is the greatest fixed point of the definition

$$\begin{array}{lll} \texttt{xs} \simeq \texttt{ys} & \Longleftrightarrow_{df} & \text{there exist } \texttt{z}, \texttt{w}, \texttt{zs}, \texttt{ws} \text{ so that } \texttt{xs} \longrightarrow \texttt{(z:zs)}, \\ & & \texttt{ys} \longrightarrow \texttt{(w:ws)}, \texttt{z} \equiv \texttt{w} \text{ and } \texttt{zs} \simeq \texttt{ws}. \end{array}$$

where the symbol $\Longleftrightarrow_{df}$ is used to mean "is defined to be".

Now, the greatest fixed point of a relation can be characterised as the union of all the post-fixed points of the relation, which in this case are called **bisimulations**. The relation $\mathcal{S}$ is a bisimulation if

$$\begin{array}{lll} \texttt{xs}\ \mathcal{S}\ \texttt{ys} & \Longrightarrow & \text{there exist } \texttt{z}, \texttt{w}, \texttt{zs}, \texttt{ws} \text{ so that } \texttt{xs} \longrightarrow \texttt{(z:zs)}, \\ & & \texttt{ys} \longrightarrow \texttt{(w:ws)}, \texttt{z} \equiv \texttt{w} \text{ and } \texttt{zs} \equiv_{\mathcal{S}} \texttt{ws}. \end{array}$$

where $\equiv_{\mathcal{S}}$ is the smallest congruence generated by the relation $\mathcal{S}$. It is now the case that

Coinduction for infinite lists.
`xs` $\simeq$ `ys` $\iff$ there exists a bisimulation $\mathcal{S}$ such that `xs` $\mathcal{S}$ `ys`.

In the next section we give an example of a proof using this coinduction principle for infinite lists.

4.8.3 An Example of Coinduction

In this section we give proof of the equality of two lists of the factorials of the natural numbers. The first is a mapping of the factorial function along the list of natural numbers

```
facMap :: [Integer]
facMap = map fac [0..]

fac :: Integer -> Integer
fac 0 = 1                                            (fac.1)
fac (n+1) = (n+1) * fac n                            (fac.2)
```

The second, `facs 0`, gives a recursive definition of the list in question. The definition uses the function `zipWith` which runs in lock step along two lists, applying a function to the elements chosen.

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _      _      = []
```

so that, for example,

```
zipWith (*) [1,1,2] [1,2,3] = [1,2,6]
```

In fact, a more general function is defined, which gives a recursive definition of the list of factorials from `n!`,

```
facs :: Integer -> [Integer]
facs n = fac n : zipWith (*) [(n+1)..] (facs n)        (facs.1)
```

To prove the equality of the two lists `facMap` and `facs 0` we first prove an auxiliary result, namely that

zipWith (*) [(n+1)..] (facs n) $\simeq$ facs (n+1)   (zipFac)

for all natural numbers `n`. In order to do this we take the relation

$\mathcal{S} \equiv \{$ `(zipWith (*) [(n+1)..] (facs n) , facs (n+1)) | n` $\in$ `Nat` $\}$

and show that it is a bisimulation. Expanding first the left hand side of a typical element we have

```
zipWith (*) [(n+1)..] (facs n)
  => zipWith (*) (n+1:[(n+2..]) (fac n : (tail (facs n)))
  => (n+1)*(fac n) : zipWith (*) [n+2..]
                                  (zipWith (*) [(n+1)..] (facs n))
  => fac (n+1) : zipWith (*) [n+2..]
                             (zipWith (*) [(n+1)..] (facs n))
```

On the right hand side we have

```
facs (n+1)
  => fac (n+1) : zipWith (*) [n+2..] (facs (n+1))
```

Now observe the two expressions. They have equal heads, and their tails are related by $\equiv_{\mathcal{S}}$ since they are applications of the function

```
zipWith (*) [(n+2)..]
```

to lists which are related by $\mathcal{S}$, namely

```
zipWith (*) [(n+1)..] (facs n) ≃ facs (n+1)
```

This establishes the result (zipFac), and the consequence that

```
facs n ≃ fac n : facs (n+1)
```

Now we prove that

```
facs n ≃ map fac [n..]
```

by showing that the relation

$\mathcal{R} \equiv \{$ `(facs n , map fac [n..]) | n` $\in$ `Nat` $\}$

is a bisimulation. Taking a typical pair, we have,

```
facs n                          map fac [n..]
  ≃ fac n : facs (n+1)             => fac n : map f [(n+1)..]
```

which establishes that $\mathcal{R}$ is a bisimulation and in particular shows that

```
facs 0 ≃ map fac [0..]
```

as we sought.

It is interesting to observe that recent work has shown that coinduction principles can be derived directly in domain theory; see [464] for more details.

4.8.4 Process Algebra

Of the many approaches to describing concurrency and non-determinism, most extend the imperative model of computation. Distinctive, therefore, are the process algebras (or process calculi) CSP [277] and CCS [393], which take a declarative model of concurrent processes. Although they differ in substantial details, their similarities outweigh their differences, and therefore the discussion here will concentrate on CCS. The use of CSP is described in detail in Chapter chap:proc.

Processes (or, rather more accurately, states of processes) are represented in CCS by expressions, with definitions of the form

```
A = (a.A + b.(B|C))
```

The process `A` is defined so that, performing the action `a`, `A` can evolve to `A`, or (`+`), performing the action `b`, it can evolve to the parallel composition `B|C`. One can see the sequencing operation, "`.`", as generalising the lazy "`:`" which appears in definitions of infinite lists like

```
natsFrom n = n : natsFrom (n+1)
```

The usual form of reasoning about CCS is equational, with equality characterised by a bisimulation relation, generalising the description in Section 4.8.2, and so one can view the lazy-stream characterisation of processes as embedding this part of functional programming in a general view of deterministic concurrency.

The set of processes in a CCS expression is fixed; in the π-calculus – which axiomatises name passing in a CCS style – processes can be created, destroyed and reconfigured, again in a declarative manner. A general introduction to the π-calculus and other action calculi is given in [394].

4.9 Strong Functional Programming

We have seen that the potential for non-termination makes program verification more complicated. Because of this there is interest in programming languages which are "strong" in the sense of providing only the means to define terminating functions.

These languages are also attractive to the implementor, since if all programs terminate however they are evaluated there is a substantially wider choice of safe evaluation strategies which can be chosen without there being a risk of introducing non-termination; this applies in particular to adopting parallel implementation strategies.

In this section we give a brief overview of various of these research directions. A general point to examine is the degree to which each approach limits a programmer's expressivity.

4.9.1 Elementary Strong Functional Programming

Turner [564] proposes a language with limited recursion and co-recursion as a terminating functional language which could be used by beginning programmers (in contrast to alternatives discussed later in this section). The language proposed will have compile-time checks for the termination of recursive definitions, along the lines of [375, 549]. The language also contains co-recursion, the dual of recursion, over co-data, such as infinite lists (the greatest fixed point of a particular type equality). An example of co-data is given by the definition of the infinite list of factorials,

```
facs = 1 : zipWith (*) [1..] facs
```

This definition is recognisable as a "productive" definition, since the recursive call to `facs` on the right hand side is protected within the constructor "`:`", and so there is a guarantee that the top-level structure of the co-datum is defined. Proof of properties of these corecursive objects is by coinduction, as discussed above.

Note the duality between these productive definitions over co-data with primitive recursive definitions over data, as exemplified by the definition of the function which gives the length of a (finite) list:

```
length (x:xs) = 1 + length xs
```

Here the recursive call to `length` is on a component of the argument, `(x:xs)`, which is contained in the application of the constructor "`:`"; thus primitive recursive definitions require at least one level of structure in their arguments whilst productive definitions give rise to at least one level of structure in their results.

The disadvantage of this approach is that it must rely on the compile-time algorithms which check for termination. It is not clear, for instance, whether the earlier definition of the `lAssoc` function is permitted in this system, and so the expressivity of the programmer is indeed limited by this approach. On the other hand, it would be possible to implement such a system as a "strong" subset of an existing language such as Haskell, and to gain the advantage of remaining in the terminating part of the language whenever possible.

4.9.2 Constructive Type Theories

Turner's language eschews the more complex dependent types of the constructive type theories of Martin-Löf and others [424, 550]. These languages are simultaneously terminating functional languages and constructive predicate logics, under the Curry/Howard Isomorphism which makes the following identifications:

Programming		**Logic**
Type		Formula
Program		Proof
Product/record type	`&`	Conjunction
Sum/union type	`\/`	Disjunction
Function type	`->`	Implication
Dependent function type	$\forall$	Universal quantifier
Dependent product type	$\exists$	Existential quantifier
...		...

in which it is possible in an integrated manner to develop programs and their proofs of correctness.

From the programming point of view, there is the addition of dependent types, which can be given by functions which return different types for different argument values: an example is the type of vectors, `Vec`, where `Vec(n)` is the type of vectors of length `n`. Predicates are constructed in a similar way, since a predicate yields different logical propositions – that is types – for different values.

Predicates (that is dependent types) can be constructed inductively as a generalisation of algebraic types. We might define the less than predicate "`<`" over `Nat` – the type of natural numbers – by saying that there are two constructors for the type:

```
ZeroLess :: (∀n::Nat)(0 < S n)
SuccLess :: (∀n::Nat)(∀n::Nat)((m < n) -> (S m < S n))
```

This approach leads to a powerful style of proof in which inductions are performed over the form of proof objects, that is the elements of types like `(m < n)`, rather than over (say) the natural numbers. This style makes proofs both shorter and more readable, since the cases in the proof reflect directly the inductive definition of the predicate, rather than being over the inductive definition of the data type, which in this case is the natural numbers.

A more expressive type system allows programmers to give more accurate types to common functions, such as function which indexes the elements of a list.

```
index :: (∀xs::[a])(∀n::Nat)((n < length xs) -> a)
```

An application of `index` has *three* arguments: a list, `xs` and a natural number `n` — as for the standard index function — and a third argument which is of type `(n < length xs)`, that is a *proof* that `n` is a legitimate index for the list in question. This extra argument becomes a **proof obligation** which must be discharged when the function is applied to elements `xs` and `n`.

The expressivity of a constructive type theory is determined by its proof-theoretic strength, so that a simple type theoretic language (without universes) would allow the definition of all functions which can be proved to be total in Peano Arithmetic, for instance. This includes most functions, except an interpreter for the language itself.

For further discussions of constructive type theories see [424, 550].

4.9.3 Algebra of Programming

The histories of functional programming and program transformation have been intertwined from their inception. Serious program manipulations are not feasible in modern imperative languages which allow aliasing, pointer and reference modifications, type casting and so forth.

More suited are current functional languages which support the definition of general operations – such as map, filter and fold over lists – as polymorphic higher-order functions. Indeed, these higher-order operators can be sufficient to define all functions of interest. This was the insight of Backus in defining FP, [27], and has been developed by a number of researchers, most notably by Bird and Meertens, [47], who are responsible for the Bird-Meertens formalism

(BMF). Their approach is to build a calculus or algebra of programs built from a fixed set of combining forms, with laws relating these combinators expressed at the function level, such as

```
map (f . g) ≡ map f . map g                               (mapComp)
```

These laws are expressed in a logic which extends the definitional equality of the programming language, and essentially equational reasoning in that logic allows transformations to be written down in a formal way. On the other hand, laws such as `(mapComp)` will themselves be proved by structural induction; for a proof of this result and many other examples of properties of list-based functions see [552].

What is the intensional reading of a law like `(mapComp)`? It shows that two traversals of a list structure, `map f . map g` is equivalent to a single traversal, `map (f . g)`; this clearly has efficiency implications. In a similar way, the fold of an associative operator into a non-empty list enjoys the property

```
foldr1 f (xs ++ ys) ≡ (foldrl f xs) ‘f‘ (foldr f ys)
                                                        (foldAssoc)
```

in the case that `xs` and `ys` are themselves non-empty. The intension of `(foldAssoc)` is dramatic, with the left hand side representing a single left-to-right traversal of a list and the right hand showing that this can be computed by means of two parallel computations over the two halves of the list.

Many of the combining forms of BMF correspond to skeletons (Chapter 13), and so the laws governing these forms will transfer to skeletons.

The most recent development of this work is Bird and de Moor's [51] in which they use the constructs of category theory to express their functional programming language. Their categorical approach means that they are able to provide general rules for equational program manipulation at a very high level. For instance, they are able to formulate in a datatype-independent way a "fusion" law by which a function is absorbed into a primitive-recursively defined function. A review of [51] which explains this work in more detail can be found in [467].

4.10 Conclusion

This chapter has given an introduction to the methods used in verifying functional programs, including references to further work in axiomatising more complex aspects of the functional paradigm. These methods can in most cases be transferred to parallel functional programs, since the proofs reflect the extensional properties of programs which are likely to be independent of the chosen evaluation strategy.

More specific ties to parallel functional programming are provided by the process model of lazy streams and its links to general process algebra. A second link is given by the combinatorial style of FP or the Bird-Meertens formalism, in which laws expressing equivalences between extensionally equal programs can be given an intensional meaning as transformations which improve efficiency or increase parallelism.

```
data IntExp = Literal Int |
              Binary Op IntExp IntExp

data Op = Add | Sub | Mul

opValue :: Op -> (Int -> Int -> Int)

eval :: IntExp -> Int

eval (Literal int) = int                                   (eval.1)
eval (Binary op ex1 ex2)
  = opValue op (eval ex1) (eval ex2)                       (eval.2)
```

```
data Code = PushLit Int |
            DoBinary Op

type Program = [Code]

compile :: IntExp -> Program

compile (Literal int)
  = [PushLit int]                                       (compile.1)
compile (Binary op ex1 ex2)
  = compile ex1 ++ compile ex2 ++ [DoBinary op]         (compile.2)

type Stack = [Int]

run :: Program -> Stack -> Stack

run [] stack
  = stack                                                   (run.1)
run (PushLit int : program) stack
  = run program (int : stack)                               (run.2)
run (DoBinary op : program) (v2:v1:stack)
  = run program (opValue op v1 v2 : stack)                  (run.3)

run _ _ = []                                                (run.4)
```

Figure 4.1. A simple interpreter and compiler for expressions

Base case

```
run (compile (Literal int) ++ program) stack
    ≡ { by (compile.1) }
run ([PushLit int] ++ program) stack
    ≡ { by definition of ++ }
run (PushLit int : program) stack
    ≡ { by (run.2) }
run program (int : stack)

run program (eval (Literal int) : stack)
    ≡ { by (eval.1) }
run program (int : stack)
```

Induction case

```
run (compile (Binary op ex1 ex2) ++ program) stack
    ≡ { by (compile.2) and associativity of ++ }
run (compile ex1 ++ compile ex2 ++ [DoBinary op] ++ program) stack
    ≡ { by the induction hypothesis for ex1 and associativity of ++ }
run (compile ex2 ++ [DoBinary op] ++ program) (eval ex1 : stack)
    ≡ { by the induction hypothesis for ex2 and associativity of ++ }
run ([DoBinary op] ++ program) (eval ex2 : eval ex1 : stack)
    ≡ { by (run.3) and definition of ++ }
run program (opValue op (eval ex1) (eval ex2) : stack)

run program (eval (Binary op ex1 ex2) : stack)
    ≡ { by (eval.2) }
run program (opValue op (eval ex1) (eval ex2) : stack)
```

Correctness theorem

```
run (compile e) []
  ≡ { substitute [] for both program and stack in (goal) }
run [] [eval e]
  ≡ { by (run.1) }
[eval e]
```

Figure 4.2. Proof of compiler correctness