HOARE LOGIC'S FOR RUN-TIME ANALYSIS OF PROGRAMS

HANNE RIIS NIELSON

DOCTOR OF PHILOSOPHY
UNIVERSITY OF EDINBURGH
1984
ABSTRACT

Hoare logic's have been used to prove partial and total correctness properties of programs. This thesis investigates how Hoare logic's can be extended to prove run-time properties as well. Such a logic should be sound and complete (in an appropriate sense) and it is also of interest whether it allows a natural formalisation of the usual informal algorithm analyses.

Three different ideas of how to formulate such a logic are studied for a language of iterative programs. It should be stressed that neither system require the program to be modified by inserting explicit operations upon a clock. All three proof systems are sound and complete (in the appropriate sense). Based on a worked example we argue that especially one of the proof systems gives rise to proofs that are rather close to the informal analyses.

This proof system is then extended to a language with nested declarations of recursive procedures with call-by-value parameters. The nesting of declarations of recursive procedures is not usually considered in Hoare logic's and motivates the introduction of an extra component into the formulas of the proof system. The proof system is shown to be sound and complete (in the appropriate sense). Example analyses substantiate the claim that this system allows a rather direct way of formalising the informal analyses of algorithms.

These proof systems express properties of the run-time by general formulas of first order logic. One may restrict the formulas to be of the form \( \text{time} \leq T \) (for some term \( T \)) in order to obtain even more natural formalisations of (some of) the informal analyses. Such a proof system is defined for the language of iterative programs and it is shown to be sound. It is not complete in the same sense as the previous proof systems although a completeness result can be obtained by defining a sufficiently strong expressiveness condition. Worked examples show that the formal proofs are very close to the informal analyses.
I wish to thank Gordon Plotkin for supervising the work reported in this thesis. In particular, it was him that suggested to investigate whether Hoare logic's could be of any use when analysing the efficiency of algorithms. I also want to thank him for ever being ready to write letters of recommendation.

Thanks are also due to Flemming Nielson, both personally and for his willingness to discuss my work.

The work presented in this thesis has been supported by a fellowship from the Science Faculty at Aarhus University, Denmark, a fellowship under the NATO Science Fellowship Programme, and a postgraduate studentship from Edinburgh University. A four month stay at Massachusetts Institute of Technology, USA, was partially supported by the Danish Natural Science Research Council.
DECLARATION

I hereby declare that this thesis has been composed by myself, that the work reported has not been presented for any university degree before, and that the work is my own. A preliminary version of Chapter 3 was accepted for the "Third Conference on Foundations of Software Technology and Theoretical Computer Science" (Bangalore, India, 1983) as "Proof Systems for Computation Time" /Ni83/.
# CONTENTS

Abstract 1
Preface 2
Declaration 3
Contents 4

## Chapter 1: Introduction

1.1 Proving the correctness of programs 10
1.2 Analysing the run-time of programs 15
1.3 Overview 22

## Chapter 2: Total correctness of while programs

2.1 The while language 38
2.2 The proof system \( \mathcal{T} \) 46
2.3 The Soundness Theorem for \( \mathcal{T} \) 54
2.4 The Completeness Theorem for \( \mathcal{T} \) 59
2.5 Concluding remarks 72

## Chapter 3: Run-time analysis of while programs

3.1 Run-time requirements for the while language 78
3.2 The proof system \( \mathcal{R} \) 85
3.3 The Soundness Theorem for \( \mathcal{R} \) 94
3.4 The Completeness Theorem for \( \mathcal{R} \) 99
3.5 Example: Bubble sorting 109
3.6 Alternative approaches 117
3.7 Concluding remarks 136

## Chapter 4: Run-time analysis of non-recursive procedure programs

4.1 The procedure language 144
4.2 The proof system \( \mathcal{R}^N \) 154
4.3 The Soundness Theorem for $\mathcal{R}^N$
4.4 The Completeness Theorem for $\mathcal{R}^N$
4.5 Example: The union-find problem
4.6 Concluding remarks

Chapter 5: Run-time analysis of recursive procedure programs
5.1 The procedure language
5.2 The proof system $\mathcal{R}^R$
5.3 The Soundness Theorem for $\mathcal{R}^R$
5.4 The Completeness Theorem for $\mathcal{R}^R$
5.5 Examples: Factorial and Merge sorting
5.6 Concluding remarks

Chapter 6: Proving upper bounds on the run-time
6.1 The proof system $\mathcal{U}$
6.2 A negative completeness result for $\mathcal{U}$
6.3 A weak completeness result for $\mathcal{U}$
6.4 Example: Dijkstra's graph algorithm
6.5 Concluding remarks

Chapter 7: Conclusions and further work

Appendix A: The completeness proof for $\mathcal{F}$
Appendix B: Proofs of properties of the non-recursive procedure language
Appendix C: Proofs of properties of the recursive procedure language

References
Index
INTRODUCTION

When designing an algorithm to solve a given problem, or when choosing between existing algorithms a number of issues are of importance. Some are vague and unformalised, for example the algorithm should be easy to understand, implement and maintain. Others are more formalisable. Perhaps the most important aspect is correctness, that is, the algorithm indeed solves the problem at hand.

In a world of limited resources it is also important that the algorithm works efficiently (or at least reasonably efficiently) when implemented on a computer. The efficiency of an algorithm can be measured in several ways; typical measures are the run-time and the storage-space requirements. Current practice, however, is to say that for an algorithm to be effective its run-time must be bounded by a polynomial in the size of its input (see for instance /Ta78/).

A programmer will often ensure that his algorithm is correct by simply running it on a sample set of data and in each case check that the correct output is produced. For larger programs a more systematic method is required. This has lead to the development of methods such as "structured programming" /DDH72/ for constructing programs that are easy to understand and to verify, and new programming languages have been introduced that make it easier to apply these methods (see for instance /Wi71/ and /LGH78/).
The need to ensure program correctness has further lead to the introduction of logical frameworks where it actually can be proved that a program is correct. Naur /Na66/ and Floyd /Fl67/ introduced independently what has come to be called the invariant assertion method. According to this method a program is represented by a flowchart and to some points in it there are associated assertions expressing properties of the program variables (it is required that every cycle of the flowchart contains at least one such point). Correctness of the program is then proved by verifying for appropriate pairs of points that if the path between them is taken and the assertion at the one end holds then the assertion at the other end will also hold.

In /Ho69/, Hoare showed how this method could be modified to reflect the structure of the programs. Hoare's approach has since received much attention. It has been extended to prove termination as well as correctness of programs (see /MaPn74/) and it has been applied to a large variety of programming constructs (for a survey, see /Ap81/). Furthermore, the method has been used to specify the semantics of programming languages (see for instance /HoWi73/) and it has been used as a tool in program design, the idea being that the programmer proves that his program is correct at the same time as he constructs it (see for instance /AlAr78/ and /Di76/).

The run-time efficiency of a program can be measured empirically by running the program on a sample set of data and in each case measuring the execution time. Just as the empirical approach is unsatisfactory for ensuring correctness of programs it is also unsatisfactory when analysing its performance. If an informed choice has to be made between two algorithms solving the same problem we
need a much more systematic analysis of their efficiency.

Several techniques have been developed for determining upper bounds on the run-time of programs. The analysis of a recursive procedure program will, for instance, normally involve construction and solution of recurrence relations. A large number of programs solving fundamental problems have been analysed using these techniques in such textbooks as "The Design and Analysis of Computer Algorithms" by Aho, Hopcroft and Ullman /AHU74/.

There are many cases where these analyses are purely syntax-directed (but there are also several important exceptions, for instance when applying the so-called book-keeping trick, see /AHU74/, /AHU82/). Aho, Hopcroft and Ullman consider in /AHU82/ a simple iterative programming language and give for each construct an informal rule for determining an upper bound on its run-time. But, as they remark, there are no fully comprehensive set of such rules. It might therefore be worthwhile to investigate to what extent the efficiency analyses found in these textbooks can be performed in a syntax-directed manner. The reasons for preferring a syntax-directed approach are similar to those given by the proponents of structural programming. There will be a small number of constructs to consider and since the analyses will have to follow the structure of the programs there will automatically be some discipline in the style of the analyses.

Enshrined in a logical formalism such as that suggested by Hoare in /Ho69/ for proving program correctness we even get a framework in which a meta-theory can be investigated. There are the logical properties of soundness and completeness: soundness intuitively expresses that what can be proved in the logical system does indeed
hold, whereas **completeness** is concerned with the applicability of the system and thereby the underlying method for analysing programs. Furthermore, the logical framework gives a basis for comparing the power of different methods and it provides a formalism with which to program automatic proof checkers.

The idea is now to extend Hoare-like proof systems to prove run-time properties and to investigate the soundness and completeness properties of such systems. However, we shall be especially interested in proof systems that give rise to proofs that, in some sense, are close to the traditional informal analyses. The reason for this is that we want to regard the informal proof as a formal proof (of the proof system) where we have not given all the details. At the "Workshop on Semantics of Programming Languages" in Göteborg Per Martin-Löf expresses it as follows (transcription of a discussion given in /Gö83 p145/):

"One shouldn't look upon a completely formal proof as something very different from, or opposed to, an informal one. A completely formal proof, where eventually each step is a step according to one of the rules of logic (...), is nothing but the informal proof that you started from but analysed in complete detail. If at every step in an informal proof you start asking 'why are you sure that B follows from A' I either fill in more steps or come to the case where I say 'this is something you have to see immediately, I cannot prove this in more detail for you'. We are then back to one of the primitive laws of logic, the informal proof has then become a formal one so there is no difference".

We shall say that a proof system allows a **natural formalisation** of the traditional informal analyses if it is possible to view the informal analyses as formal proofs in the proof system where we have omitted some of the details. This is a rather vague definition of a property of a proof system and consequently we shall only discuss it on the basis of examples (mainly chosen from /AHU74/ and /AHU82/). The main examples are: the bubble sorting algorithm, an algorithm
solving the union-find problem, the merge-sort algorithm and Dijkstra's graph algorithm.

The rest of this chapter is organised as follows. In Section 1.1 we review Hoare's method for proving program correctness. We illustrate the method for a simple iterative language and discuss informally the theoretical properties of soundness and completeness. The subject of Section 1.2 is run-time analysis. Here we review the general framework and give a set of informal rules for analysing the run-time of iterative programs. There has already been some attempts towards using formal systems in run-time analysis and in Section 1.2 we review some of them. Finally in Section 1.3 we give an overview of this thesis.

The reader is assumed to be acquainted with the usual informal analysis of programs as presented in for instance /AHU82/. The general idea of a proof in a proof system (as for instance Hoare logic /Ho69/) should be known; the theoretical concepts will be explained when needed.

1.1 PROVING THE CORRECTNESS OF PROGRAMS

Hoare considers in /Ho69/ a simple iterative language and gives a proof system for reasoning about the correctness of its programs. The basic formulas of the proof system have the form P[c]Q (usually written \{P\}c\{Q\}) where c is a program and P and Q are formulas of some assertion language. The idea is that the formula P, called the pre-condition, expresses a property of the initial values of the program variables of c, whereas the formula Q, called the post-condition, expresses a property of the final values. The formula P[c]Q may be interpreted as saying that if P holds before c is executed then upon
termination \( Q \) will hold. For instance, we may write
\[
x = x_0 \text{[WHILE } x \geq y \text{ DO } x := x - y \text{]} (\exists q. x_0 = q' y + x) \land x < y
\]
to express that the program \text{WHILE } x \geq y \text{ DO } x := x - y \text{ computes the remainder of } x \text{ upon division by } y.

**HOARE'S PROOF SYSTEM**

For a simple language with the three constructs of assignment, composition and iteration, Hoare gives the following set \( \mathcal{H} \) of axioms and rules /Ho69/:

The proof system \( \mathcal{H} \):

\[
\begin{align*}
/\text{ass-} & / & P_{x}^{e}[x := e]P \\
/; & / & P[c_{1}]Q, Q[c_{2}]R \\
& & P[c_{1}; c_{2}]R \\
/\text{WHILE-} & / & P \left[ \text{WHILE } b \text{ DO } c \right] P_{a \Rightarrow b} \\
/\text{cons-} & / & P \rightarrow P', P'[c]Q', Q' \rightarrow Q \\
& & P[c]Q
\end{align*}
\]

Here \( P_{x}^{e} \) stands for the usual substitution of the expression \( e \) for all free occurrences of the variable \( x \) in \( P \).

We can, using this proof system, prove the formula
\[
x = x_0 \text{[WHILE } x \geq y \text{ DO } x := x - y \text{]} (\exists q. x_0 = q' y + x) \land x < y
\]
Let \( P \) be the assertion \( \exists q. x_0 = q' y + x \). Then from /\text{ass-} / we get
\[
P_{x}^{x-y}[x := x - y]P
\]
and since \( P_{a \Rightarrow y} \rightarrow P_{x}^{x-y} \) holds, that is \((\exists q. x_0 = q' y + x) \land x < y \rightarrow \exists q. x_0 = q' y + (x - y)\)
we get using /cons-\(\mathcal{K}\)/ a proof of

\[ P \land x \equiv y [x := x - y] \mathcal{P} \]

An application of /WHILE-\(\mathcal{K}\)/ now gives

\[ P [\text{WHILE } x \geq y \text{ DO } x := x - y] \mathcal{P} \land (x \equiv y) \]

and the required proof is obtained by using /cons-\(\mathcal{K}\)/ with \(x = x_0 \rightarrow P\), that is, \(x = x_0 \rightarrow \exists q. x_0 = q \cdot y + x\).

The assertions are usually formulas of some first-order language built on top of the expression language of the programming language. The formula \((\exists q. x_0 = q \cdot y + x) \land x \leq y\) is a typical example of a formula in the assertion language. It contains both program variables (\(x\) and \(y\)) and so-called logical variables (\(x_0\) and \(q\)). The formula \(x \leq y\) will also be in the expression language but \(\exists q. x_0 = q \cdot y + x\) will certainly not be.

The rule of consequence /cons-\(\mathcal{K}\)/ shows that the assertion language is an integral part of the proof system. The usual approach is to restrict the use of the rule to cases where both \(P \rightarrow P'\) and \(Q' \rightarrow Q\) are true in a specific interpretation for the assertion language. This is also the approach we shall use. Another possibility would be to extend the proof system with axioms and rules for the assertion language; this approach has been studied in for instance /BeTu80/ and /BeTu81/.

**Soundness and Completeness Properties**

Above we saw how to prove the formula

\[ x = x_0 [\text{WHILE } x \equiv y \text{ DO } x := x - y] (\exists q. x_0 = q \cdot y + x) \land x \leq y \]

in the proof system \(\mathcal{K}\). The whole idea is now that we want to interpret this as: the program WHILE \(x \geq y\) DO \(x := x - y\) computes the remainder of \(x\).
upon division by $y$. The soundness of the proof system allows us to do that.

Informally, **soundness** means that everything that can be proved using the axioms and rules of the proof system (and the true formulas of the assertion language) does indeed hold. Intuitively, the proof system above satisfies this. Given a rigorous definition of the semantics of the programming language it can be formally proved that the proof system is sound for every interpretation of the assertion language (see for instance /Co78/).

The converse property to soundness is **completeness** and is about the applicability of the proof system: any property that holds for a program can also be proved to hold in the proof system. Wand shows in /Wa78/ that Hoare's proof system cannot be complete in that sense: for a given assertion language and a given interpretation for it he constructs a formula $P[c]Q$ that indeed does hold but cannot be proved in the proof system.

Intuitively, the reason is that the assertion language contains too few formulas to be able to express the required intermediate assertions in a proof. To overcome that problem, Cook introduces in /Co78/ an **expressiveness condition** on the assertion language and its interpretation, and he proves that when it is fulfilled the proof system is indeed complete.

Cook's expressiveness condition requires that the **strongest post-conditions** are expressible in the assertion language, that is, for each program $c$ and formula $P$ there must be another formula $Q$ that will hold for all and only those states that are reachable by execution of $c$ from a state where $P$ holds. This, rather technical
requirement, is satisfied by a large class of assertion languages and interpretations, for instance ordinary number theory with its standard model.

Cook shows that expressiveness is a sufficient condition to ensure completeness of the proof system but, as Bergstra and Tucker show in /BeTu80/, it is not a necessary condition. The question of how restrictive the assumption is has been studied in several papers, for instance /C179/, /Li77/ and /BeTu80/.

**PROVING TERMINATION**

Hoare's proof system given above can be used to prove *partial correctness* properties of programs: we can prove that if the pre-condition $P$ of the formula $P[c]Q$ holds when the execution of $c$ is started and if the computation terminates then the post-condition $Q$ will hold for the final values of the variables. One cannot argue about termination of the program within the formalism.

Manna and Pnueli have in /MaPn74/ extended Hoare's method to prove *total correctness* of programs, that is, to prove termination as well as correctness. Other extensions have been suggested by Sokolowski /So77a/, Harel /Ha79/ and Aczel /Ac82/; the main difference being concerned with how termination of the while loop is ensured. We shall not give the details here but merely note that they exist. In Chapter 2 we shall develop a proof system for total correctness using ideas from /MaPn74/, /Ha79/ and /Ac82/.
1.2 ANALYSING THE RUN-TIME OF PROGRAMS

When analysing the run-time of an algorithm it will always be with respect to some mathematical model specifying the time requirements of the primitive operations. The Random Access Machine /AHU74/ is an example of a model that in many cases is an appropriate abstraction of the real computer. In some analyses one will focus upon certain critical operations; for instance, in sorting problems the comparison of data may be the overall most expensive operation and one may therefore employ a computational model where all other operations are free.

The run-time of a program depends on its input. By the exact time complexity of a program $c$ we shall understand the (partial) function $T_c$ that to each input associates the corresponding run-time of $c$. Often one is interested in the worst-case time complexity of the program. It is defined to be the maximum, over all inputs on that given size, of the run-time on that given input, that is, it is a (partial) function $W_c$ that to each size $n$ of input associates the number

$$\max\{T_c(s) | s \text{ is an input to } c \text{ of size } n\}$$

Finally, one might be interested in the order of magnitude of the worst-case time complexity. If $f$ is a function mapping natural numbers to natural numbers then we say that $W_c$ is of order of magnitude $f$ if there are constants $n_0$ and $k$ such that

$$\text{if } n \geq n_0 \text{ then } W_c(n) \leq k f(n)$$

We write "$W_c(n)$ is $\Theta(f(n))$".

As an example consider again the program WHILE $x \geq y$ DO $x := x - y$. Assume that the test $x \geq y$ as well as the assignment $x := x - y$ requires
one unit of time. Then the exact time complexity of the program is $2^\left\lfloor x/y \right\rfloor + 1$ (assuming $y > 0$). If the size of input is given by the value of $x$ then the worst-case time complexity is $2^x + 1$ and it is of order of magnitude $x$.

In the literature there are a lot of examples showing how to prove run-time properties of various algorithms but there are also a few suggestions of principles that enable these analyses to be performed more or less automatically. In the rest of this section we review some of these suggestions. First we list some informal rules for analysing the order of magnitude of the worst-case time complexity of simple iterative programs suggested by Aho, Hopcroft and Ullman in /AHU82/. Then we review some work on the so-called loop programs. Meyer and Ritchie have in /MeRi67/ studied the run-time of these programs and their work has been continued by Adachi, Kasai and Moriya in /KaAd80/ and /AKM79/. Finally, we shall review two approaches based on a logical framework. The method of programmed counters has been suggested by several authors, for instance Wegbreit in /We76/ and Knuth in /Kn73/. A method using predicate transformers has been suggested by Shaw in /Sh79/.

There have also been suggestions for formalising average-case analyses of run-time and the main references are here /We76/, /Ra80/ and /Ko83/. In general average-case analyses tend to be mathematically more complicated than worst-case analyses and since we argue that a satisfactory formalisation of the latter has not yet been obtained we do not consider average-case analyses further in this thesis.
Analysing the run-time of a program can be a difficult mathematical problem but in many cases a few basic rules are sufficient. Aho, Hopcroft and Ullman give in /AHU82 p23/ the following set of rules for analysing the run-time requirements of the programs in the simple iterative language considered earlier; more precisely, the rules will give the order of magnitude of the worst-case time complexity of the programs.

**Assignment:** the run-time is $O(1)$, that is, it is bounded by a constant

**Composition:** the run-time is, to within a constant factor, the largest of the run-times of the statements

**Iteration:** the run-time of the loop is the sum, over all times round the loop, of the time to execute the body and the time for evaluating the condition (usually $O(1)$). Often this time is, neglecting constant factors, the product of the number of times the loop is executed and the maximal run-time for the body of the loop.

As an example consider again the program WHILE $x\leq y$ DO $x:=x-y$.

According to the assignment rule the statement $x:=x-y$ requires time $O(1)$. The body of the loop is executed $\lfloor x/y \rfloor$ times so the iteration rule gives that the runtime of the program is $O(\lfloor x/y \rfloor)$ (which in turn is $O(x)$).

**Loop Programs**

The language of loop programs is obtained from the simple iterative language considered earlier by replacing the while construct with a construct LOOP $x$ DO $c$ that resembles the for-loop of the programming language Fortran and special cases of the for-loop of Algol-like languages. Furthermore, the expressions of the loop language are very
restricted (only 0, x, x+1 and x-1 are allowed). The following is an example of a loop program that, semantically, corresponds to the program \( \text{WHILE } x' \geq y \text{ DO } x := x - y \) (if \( y > 0 \)):

\[
\text{LOOP } x \text{ DO (z := x; LOOP } y \text{ DO z := z - 1; LOOP } z \text{ DO x := z)}
\]

In contrast to the simple iterative programs, a loop program will terminate on every input. The problem is now to find upper bounds on the run-time of every loop program.

Meyer and Ritchie define in /MeRi67/ an infinite hierarchy

\[
L_0 < L_1 < L_2 < \ldots
\]

of loop programs: \( L_n \) is the set of loop programs with the maximal depth of loop nesting being \( n \). For each program in \( L_n \) they can determine a function bounding its run-time. For a program in \( L_2 \) (as that given above) this bound is a \( k \)-fold exponential function - the constant \( k \) is determined syntactically from the program.

For programs as the above example this bound on the run-time seems to be rather high and therefore uninformative. Kasai and Adachi show in /KaAd80/ how more precise upper bounds can be obtained for the so-called simple loop programs. Their idea is as follows. The variable \( x \) occurring in the statement \( \text{LOOP } x \text{ DO } c \) is called a control variable. Each loop program \( c \) defines a relation \( c < \) on the control variables occurring in it:

\[
x < c_y \text{ if and only if } c \text{ contains a statement } \text{LOOP } x \text{ DO } c',
\]

where \( c' \) contains the statement \( y := y + 1 \)

A simple loop program is now a loop program \( c \) where no chain

\[
x_1 < c_2 < \ldots < c_k
\]

contains repetitions (that is \( x_i \neq x_j \) for \( i \neq j \)).

The run-time of a simple loop program \( c \) is bounded by a term
\[ k_1 \cdot (\max\{x_1, \ldots, x_k\})^{\text{lc}(c)} + k_2 \]

where \( k_1 \) and \( k_2 \) are constants, \( x_1, \ldots, x_k \) are the control variables occurring in \( c \) and \( \text{lc}(c) \) is some constant defined from the syntax of \( c \) and called its loop complexity. For the example program mentioned earlier this approach gives an upper bound of \( k_1 \cdot (\max\{x,y,z\})^4 + k_2 \).

This bound may also seem quite high but is certainly more informative than that of /MeRi67/.

Adachi, Kasai and Moriya improve these results in /AKM80/. The idea is now to apply a certain algorithm to a transformed version of the program. In the program there are inserted statements updating the time used so far; the example program considered earlier is thus transformed into the following program:

```
time:=0; LOOP x DO (z:=x; time:=time+1;
                LOOP y DO (z:=z-1; time:=time+1);
                LOOP z DO (x:=z; time:=time+1))
```

The variable \text{time} is regarded as a control variable and for each control variable the algorithm will construct a term bounding its value. We shall not give the details of the rather complicated algorithm here but merely note that it can be used to prove an \( x^4(y+x+1) \) upper bound on the run-time of the loop program considered in the beginning of this subsection.

\textbf{THE METHOD OF PROGRAMMED COUNTERS}

The idea in this approach is to transform the program so that it counts the time used so far in a special variable and then use Hoare's proof system (see Section 1.1) to prove some property of the counter in the transformed program. This property will then hold for the run-
time of the original program. For instance we can prove that $2^t x + 1$ is an upper bound of the worst-case time complexity of the program

WHILE $x \geq y$ DO $x := x - y$ by proving the formula

$$x = x_0 \left[ \begin{array}{c}
\text{time} := 0; \\
\text{WHILE } x \geq y \text{ DO (time := time + 1; x := x - y; time := time + 1)}
\end{array} \right]$$

using Hoare's proof system $\mathcal{H}$ given in Section 1.1.

This approach has been suggested by Wegbreit in /We76/ and it also occurs in /Kn73/. The idea has been used by Luckham and Suzuki in /LuSu77/ when proving termination of programs. As we have seen the idea of proving a property of a program by transforming it to count the time used so far also occurs in the work of Adachi, Kasai and Moriya /AKM80/.

**Shaw's method**

As Shaw notes in /Sh79/ the program transformation employed in the above approach using programmed counters is mechanical but tedious and furthermore it is subject to human error. So she suggests extending the proof rules of the programming language to perform the accounting of time automatically.

A pseudo variable _time_ will count the time used so far in the execution of the program, and it will implicitly be incremented each time an operation has been performed. The variable will occur in formulas expressing run-time properties just as the program variables do. A predicate transformer for each of the constructs of the programming language will increment the value of the variable _time_ in addition to performing the transformation appropriate for the con-
struct.

For each program $c$ and formula $R$ of the assertion language there is a formula $cp(c, R)$ called the cost-precondition of $c$ and $R$. The idea is that $R$ expresses a property of the run-time for the part of the program preceding the program $c$ and for $c$ itself. The variable $time$ stands for the time used so far, thus

$$time + \text{(time required for the computation following $c$)} = \text{(total time for the program)}$$

The formula $cp(c, R)$ will then express a property of the time used before the execution of $c$ is started, so, in some sense we are pulling the run-time information backwards through the program.

For the iterative language considered earlier we have the following rules (simplified versions of those of /Sh79/):

- $cp(x := e, R) \implies R(e, time + 1)$
- $cp(c_1; c_2, R) \implies cp(c_1, cp(c_2, R))$
- $cp(\text{WHILE } b \text{ DO } c, R) \implies V_{i=0}^\infty H_i$

where $H_0 \implies bAR\text{, time} + 1$

and $H_{i+1} \implies bACp(c, H_i)\text{, time} + 1 \quad (i \geq 0)$

We have here assumed that the evaluation of terms and boolean expressions takes one unit of time.

Using these rules we can prove that the program WHILE $x \geq y$ DO $x := x - y$ has worst-case time complexity $2 \times n + 1$. More precisely, we prove that

$$x = x_0 \land y \geq 0 \land time = 0 \implies cp(\text{WHILE } x \geq y \text{ DO } x := x - y, time \leq 2 \times x_0 + 1)$$

Intuitively, the verification of this formula consists in propagating
the formula $t_{\text{ime}} = 2^2 \times x_0 + 1$ from the end of the program towards its beginning.

1.3 OVERVIEW

We claim that the method of programmed counters (as well as those for loop programs) reviewed in the previous section is not a satisfactory basis for the formalisation of the informal run-time analyses found in textbooks as for instance /AHU82/. The most obvious obstacle is the program transformation. Although it can easily be performed mechanically it is against the spirit of the informal analyses of /AHU82/. Many of these follow the rules mentioned in Section 1.2, where the run-time requirement of a composite construct is given in terms of those for its components as they are actually written.

The method based on predicate transformers avoids the program transformation by giving a set of rules especially designed to handle run-time properties. But again, we claim that it is not appropriate as a basis for a formalisation of the informal analyses of /AHU82/. First, we note that in algorithm analysis it is implicitly assumed that the complexity of a program is expressed in terms of its input and such a discipline is not imposed by this approach. Secondly, another problem can be illustrated by considering the composition construct of the programming language. The informal rule will inspect the run-time properties for the two components and then find a property of the total run-time. The idea behind Shaw's rule is different. Here the assertion about the run-time is "pulled backwards" through the components and for instance a comparison of the run-time requirements of the two components seems more or less impossible. This will be discussed further in Section 3.6.
The two logical approaches sketched in the previous section both have the idea of a global time and this seems to make it difficult to talk about the run-time of a subprogram. However, the informal rules of /AHUB82/ mentioned in Section 1.2 refer to the run-time of the components of composite constructs. Our first idea is therefore to avoid the global time count and instead keep track of the run-time for each subprogram separately - or, put in another way, to keep a global clock but to allow one to talk about time differences.

With this idea in mind we shall extend a Hoare-like proof system to prove run-time properties. In the rest of this section we shall describe in more detail how this is carried out and we shall motivate some of the important choices. The main body of this thesis consists of five chapters and they will be described in turn below.

**Total correctness of while programs**

Chapter 2 is devoted to a presentation of a proof system for proving total correctness of while programs. The proof systems for run-time analysis developed in the remaining chapters will all be based upon this proof system. The proof system is chosen such that the run-time proof systems can be formulated as conveniently as possible. Let us motivate our choice.

A property of the run-time of a program, say \( c \), can be thought of as a relation between the values of its input variables and the corresponding run-time. An example is an upper bound on the run-time: we shall write it as \( \text{time} \leq T \) where \( \text{time} \) is a (logical) variable denoting the run-time of \( c \) and \( T \) is a term where only the input variables of \( c \) occur free.
Let us now discuss what a proof rule for the composite construct \( c_1;c_2 \) should be. For the sake of simplicity assume that we have given upper bound properties \( \text{time}_1 \leq T_1 \) and \( \text{time}_2 \leq T_2 \) for the run-time of \( c_1 \) and \( c_2 \), respectively. The run-time of \( c_1;c_2 \) is \( \text{time}_1 + \text{time}_2 \) but we cannot expect that \( T_1 + T_2 \) is an upper bound. The reason is simply that the references of the free variables of \( T_2 \) becomes wrong: \( \text{time}_2 \leq T_2 \) expresses a relation between the values of the input variables of \( c_2 \) and the corresponding run-time, whereas a property of the run-time of \( c_1;c_2 \) has to be a relation between the initial values of the input variables for that program and the corresponding run-time. In many cases \( c_1 \) might be such that the variables occurring free in \( T_2 \) are not changed (and then \( \text{time}_1 + \text{time}_2 \leq T_1 + T_2 \) will indeed hold) but it is not the case in general.

A solution to the problem is, of course, to keep track of how the values of the variables are modified by \( c_1 \). This information can then be used to transform the term \( T_2 \) into another term \( T' \) with the property that when it is evaluated in the initial state of \( c_1 \) then it gives the same value as \( T_2 \) evaluated in the final state of \( c_1 \) (which is the initial state of \( c_2 \)). Then \( \text{time}_1 + \text{time}_2 \leq T_1 + T_2 \) will hold for the run-time of \( c_1;c_2 \).

Hoare's proof system for partial correctness properties (see Section 1.1) can easily be used to prove relations between initial and final values of variables. The idea is to take a so-called snapshot of the initial values of the variables in the pre-condition, as for instance in the formula

\[
x = x_0 [\text{WHILE } x \geq y \text{ DO } x := x - y ] (\exists q. x_0 = q \cdot y + x) \land x < y
\]

considered earlier.
Many applications of Hoare-like proof systems show that one quite often will express relationships between the initial and the final values of the variables - and this is not surprising since it is simply one way of describing the effect of executing a program. Manna and Pnueli suggest in /MaPn74/ a version of Hoare's system where the post-condition of a formula always expresses a relation between the initial and the final values of the variables rather than just a property of their final values. The idea has also been taken up by Jones in /Jo80/.

The proof system we shall develop in Chapter 2 will have formulas where the post-conditions express relations between the initial and the final values of the variables. Pictorially, we express it as follows:

The axioms and rules of the proof system will tell how these relationships are defined in terms of each other for the various constructs. The rule for composition of two programs $c_1$ and $c_2$ will, for instance, act as follows. Given that we have the relationships for the programs $c_1$ and $c_2$ marked \( \Downarrow \text{ and } \Uparrow \) on the figure below we can conclude that the relationship marked \( \Rightarrow \) will hold for the program $c_1; c_2$. 

![Diagram](image-url)
The proof system we shall construct will be a proof system for **total correctness** properties rather than partial correctness properties. This means that we in the later chapters will develop proof systems that allow us to prove properties stating that the program will terminate and its run-time will satisfy such and such a property. An alternative would, of course, be to extend a proof system for partial correctness to prove run-time properties as well. However, the traditional informal analyses of for instance /AHU82/ do not consider this kind of properties.

In Section 2.1 we present the language of while programs in detail and we give a precise definition of its semantics using an operational approach. The proof system for total correctness is presented in Section 2.2 and it is proved to be sound (in Section 2.3) and complete in the sense of Cook (in Section 2.4). Finally, in Section 2.5 we compare the proof system with some of the existing ones.

**RUN-TIME ANALYSIS OF WHILE PROGRAMS**

In Chapter 3 we shall extend the total correctness proof system to prove run-time properties as well. Before doing so it is important to decide what we mean by "a property of the run-time of a program" and
thereby what sort of run-time properties we want to formalise by the proof system. As mentioned earlier we shall view a **property of the run-time** of a program as a relation between the initial state and the corresponding run-time. An upper bound or lower bound of the run-time can easily be viewed in that way and so can (an upper bound of) the worst-case time complexity of a program. (However, there are certain limitations to this approach; for instance the expected time complexity cannot be viewed as a relation between a single state and the corresponding run-time.)

Given this notion of a property of the run-time of a program we can extend the proof system for total correctness to prove run-time properties in several ways. In the **direct style approach** the idea is to extend each formula of the proof system to express a relation between the initial state and the corresponding run-time. Pictorially, we shall express it as follows:

![Diagram](image)

The axioms and rules of such a proof system will be extensions of those of the total correctness proof system. To illustrate this further let us again consider the rule for composition of two programs $c_1$ and $c_2$. Given that we have the relationships marked and on the figure below between the initial states and the run-times of $c_1$ and $c_2$, respectively, we shall find a relationship between
the initial state of $c_1; c_2$ and the total run-time of $c_1; c_2$.

The idea is to use the relationship between the initial and the final state of $c_1$ to get a relation between the run-time of $c_2$ and the initial state of $c_1$ (rather than $c_2$). On the figure below it is marked \( \Rightarrow \Rightarrow \Rightarrow \Rightarrow \). A special "adding mechanism" will then put the relationships marked \( \Rightarrow \Rightarrow \Rightarrow \Rightarrow \) and \( \Rightarrow \Rightarrow \Rightarrow \Rightarrow \) together and we obtain the required relationship (marked \( \Rightarrow \Rightarrow \Rightarrow \Rightarrow \)).

The main proof system developed in Chapter 3 is based upon these ideas. The proof system is proved to be both sound and complete in the appropriate senses so it has the desired theoretical properties. As mentioned earlier we shall be interested in the formalisation of the existing informal analyses of algorithms as proofs in the proof system and furthermore we want this formalisation to be "as natural as
possible". We shall therefore consider a number of worked examples in this thesis and the first of these, the bubble sorting algorithm, is considered in Chapter 3. We give a formal proof of the \( \Theta(n^2) \) upper bound on the run-time of this algorithm and compare the proof with that of the traditional informal analysis.

At the first glance it seems to be the case that we do a lot of extra work in the formal proof but, as in the formalisation of the informal correctness proofs, most of these deductions correspond to deductions in the informal analysis that are simply ignored because they are regarded as more or less "obvious". More importantly, the informal and the formal proof proceed in essentially the same way: given run-time properties for the constituents of a statement we find a run-time property of the complete statement. However, the way these run-time properties are obtained are different, especially for the while loops. In the informal analysis we are summing a series whereas in the formal proof it turns out that we have to find a formula satisfying certain properties. These properties will be satisfied by choosing the formula to express that the run-time is the sum of the series. In Chapter 6 we shall return to a discussion of these matters.

The idea behind the approach of Shaw /Sh79/ described in Section 1.2 is rather different from that presented above in that it views a program as a transformer of properties of the run-time "used so far". In Chapter 3 we shall also show how the total correctness proof system of Chapter 2 can be extended to prove run-time properties using these ideas. The resulting proof system turns out to be essentially as powerful as the direct style proof system, at least from a theoretical point of view. However, based on the bubble sorting algorithm we argue that it does not lead to natural formalisations of informal ana-
lyses. Furthermore, we shall exhibit a so-called continuation style proof system that in some sense is dual to that of /Sh79/. Again we claim that the direct style proof system should be preferred.

In Section 3.1 we extend the operational semantics of the while language given in Section 2.1 to specify the run-time requirements of the programs as well as their semantics. The proof system for run-time analysis is presented in Section 3.2 and it is proved to be sound and complete (in the appropriate senses) in the sections 3.3 and 3.4, respectively. Section 3.5 contains a worked example, the bubble sorting algorithm. Some alternative proof systems are discussed in Section 3.6 and finally, Section 3.7 contains some concluding remarks.

**RUN-TIME ANALYSIS OF NON-RECURSIVE PROCEDURE PROGRAMS**

The proof system developed in Chapter 3 is far from sufficient if one wants to formalise the informal run-time analyses of a larger class of the algorithms considered in for instance /AHU82/. Maybe the most obvious weakness of the proof system is that it only applies to while programs. In the chapters 4 and 5 we shall extend the language to contain some further interesting programming constructs.

The language considered in Chapter 4 introduces three new statements in the language for

- declaration of local initialised variables
- declaration of local (non-recursive) procedures with call-by-value and call-by-variable parameters
- call of procedures

We shall extend the proof system of Chapter 3 to prove run-time pro-
properties of programs in this language and we shall prove soundness and completeness results that are similar to those obtained in Chapter 3.

The informal analysis of run-time properties of programs in the non-recursive procedure language is fairly straightforward. Since the procedures cannot call each other recursively in a given program they can be ordered so that each procedure can only call procedures preceding it in the ordering. The idea is then first to analyse the body of the procedure that makes no calls of other procedures, and then the body of the succeeding one and so on.

As an example of the use of the proof system we shall analyse (one of) the algorithms solving the so-called union-find problem, see for instance /AHU82/. The informal analysis gives that the run-time of the algorithm is \( O(n \cdot \log(n)) \). We have at least three strategies we may attempt in order to construct a proof of this result in the proof system.

One possibility is to proceed in a bottom-up manner and use the axioms and rules of the proof system to prove run-time properties of larger subprograms from those of the smaller ones. The hope is then that we will end up with a proof of a \( k \cdot n \cdot \log(n) \) upper bound on the run-time of the complete program (for some constant \( k \)) and we have then completed the proof.

Another possibility will be to guess a constant \( k \) and then verify in a top-down manner that \( k \cdot n \cdot \log(n) \) is an upper bound on the run-time of the program. This means that we use the rules of the proof system to split the problem of finding a proof for a run-time property of a composite program into smaller problems for the various components.
The third possibility, and the one that is used for the union-find algorithm, is to forget everything about the $\Theta(n \cdot \log(n))$ bound for a moment and then look for conditions on a term, say $T(n)$, that will ensure that it is an upper bound on the run-time of the algorithm. More precisely, we shall construct a "proof" for the upper bound $T(n)$ in the proof system without knowing exactly what $T(n)$ is. However, in order to get that "proof" we have to make some assumptions about $T(n)$ and if they are satisfied by some given term - for instance $k' n \cdot \log(n)$ for some constant $k$ - then we will automatically have a proof showing that this term is an upper bound on the run-time of the program. The construction of the "proof" will often be a mixture of top-down and bottom-up development and it may be useful to invent other "unknown terms" in order to get a smooth development. As we shall see in later examples this approach seems in many cases to lead to natural formalisations of informal analyses.

In Section 4.1 we present the language of non-recursive procedure programs, its semantics and its run-time requirements. The proof system for proving run-time properties of programs in the language is given in Section 4.2 and its soundness and completeness properties are considered in the two sections 4.3 and 4.4, respectively. Section 4.5 shows how an algorithm solving the union-find problem can be analysed in the proof system. In Section 4.6 we give some concluding remarks.

**RUN-TIME ANALYSIS OF RECURSIVE PROCEDURE PROGRAMS**

Recursion is an important and very general technique for algorithm construction. Often an algorithm can be stated much simpler using recursion than without it. Furthermore, there exists programming strategies such as divide-and-conquer and dynamic programming that are
based on recursion and in many cases give efficient algorithms. In Chapter 5 we shall therefore consider a recursive variant of the language of Chapter 4 and we shall develop a sound and complete (in appropriate senses) proof system for analysing run-time properties of programs in this language.

The (informal) run-time analysis of recursive programs is rather different from that of non-recursive programs and it normally consists of construction and solution of recurrence relations. Given a recursive program the recurrence relation is obtained as follows: to each (recursive) procedure there is associated an unknown time-function $T(n)$ where $n$ is the size of the argument to the procedure. Then the body of the procedure is analysed and this will give rise to an equation for $T(n)$ in terms of $T(k)$ for various values of $k$ (corresponding to the recursive calls of the procedure). This equation can then be solved using various techniques (see for instance /AHU82/).

As an example let us consider one of the standard divide-and-conquer algorithms, the merge sort algorithm:

```plaintext
PROC merge-sort(VAL i, VAL m) IS

PROC merge(VAL i, VAL m) IS ...

IN IF m=1 THEN m:=1
ELSE (CALL merge-sort(i,m/2);
    CALL merge-sort(i+m/2,m/2);
    CALL merge(i,m/2))

IN CALL merge-sort(1,length(l))
```

The procedure merge-sort will sort the part $l[i+1],...,l[i+m]$ of the array $l$ by calling itself recursively on the two parts $l[i+1],...,l[i+m/2]$ and $l[i+m/2+1],...,l[i+m]$. The procedure call CALL merge(i,m/2)
will merge these two sorted sublists.

Let now $T(m)$ be an upper bound for the run-time of the procedure merge-sort. An informal analysis will then give rise to the following recurrence relation (see for instance /AHU82 p295/):

$$(S) \quad T(m) = \begin{cases} 
  c_1 & \text{if } m = 1 \\
  2T(m/2) + c_2 \cdot m & \text{if } m > 1
\end{cases}$$

Here $c_1$ represents the constant number of steps taken when the length of the list to be sorted is one. In the case where it contains more than one element the time requirements can be divided into two parts. First there are the two recursive calls of merge-sort to sort lists of length $m/2$ each taking time $T(m/2)$, thus we get the term $2T(m/2)$ in $(S)$. Secondly, time is required for the test $m=1$ in the procedure body and for the call of the procedure merge. The test requires constant time and the procedure call requires time proportional to $m$. The constant $c_2$ is chosen such that $c_2 \cdot m$ is an upper bound on the time required for these operations. This explains the form of the recurrence relation $(S)$. Assuming that $m$ is a power of two we get that

$$T(m) = (c_1 + c_2) \cdot m \cdot \log(m) + c_1$$

is a solution to $(S)$.

In Chapter 5 we develop a proof system that can be used to analyse the run-time requirements of a program as that above. Although the language of Chapter 5 can be viewed as a simple modification of that of Chapter 4 the proof system turns out to be considerably more complicated. This is not only because the run-time analysis is more involved but is mainly inherited from the underlying proof system for total correctness.
We prove that the proof system for run-time analysis of recursive procedure programs is both sound and complete (in the appropriate senses). As for the pragmatic issues, the interesting question is whether we obtain some sort of recurrence relation when analysing an algorithm in the formal system. An analysis of the merge sort algorithm above shows that this is indeed the case (at least if we adopt an appropriate proof strategy).

In Section 5.1 we present the semantics and run-time requirements of the recursive procedure language. The proof system for proving run-time properties of programs in this language is given in Section 5.2 and it is proved to be sound and complete (in the appropriate senses) in the sections 5.3 and 5.4, respectively. A couple of example analyses are presented in Section 5.5. Finally, in Section 5.6 we show how the proof system for run-time analysis specifies a proof system for total correctness. This proof system differs from other Hoare-like proof systems for total correctness in that it allows nested declarations of recursive procedures. Corollaries to the results of the sections 5.3 and 5.4 show that also this proof system is sound and complete (in the appropriate senses).

**Proving upper bounds on run-time**

In the two chapters 4 and 5 we extend the while language to contain some important programming constructs and we develop proof systems for proving run-time properties of programs in these languages. In Chapter 6 we return to the language of while programs and the idea is now to improve the previous proof system given in Chapter 3 so that we can do a better job of formalising the informal run-time analyses of programs in this language.
As mentioned earlier we are mainly interested in analyses of the order of magnitude of the worst-case time complexity of the algorithms. These analyses will involve some notion of the size of input to a program as well as calculations with "orders of magnitudes". However, we are also left with problems that in some sense are more basic. The proof system of Chapter 3 does not give us the formalisation of the informal proof of the bubble sorting algorithm that we might want. The main problem is here to formalise the informal analysis of the (outer-most) while loop in a satisfactory way.

As a step towards a proof system for proving the order of magnitude of the worst-case time complexity of programs we develop in Chapter 6 a proof system for proving upper bounds on the run-time of programs and we investigate its theoretical as well as its pragmatic properties.

Formally, the proof system for upper bounds is derived from the proof system of Chapter 3 and this means that it is sound in (essentially) the same sense. It turns out that the new proof system is not complete in exactly the same sense as the previous ones but under crude assumptions we can obtain a completeness result.

Using the upper bound proof system we obtain a quite satisfactory formalisation of the informal analysis of the bubble sorting program. As mentioned earlier, the proof system of Chapter 3 requires us to find some property of the run-time of the (outer) while loop that satisfies certain conditions. The upper bound proof system requires that we find a term solving a simple recurrence relation. In fact, the recurrence relation is a straightforward reformulation of the summing series of the informal analysis.

The informal analysis of the (outer) while loop in the bubble sorting
algorithm uses the first part of the informal rule stated in Section 1.2: "the run-time of the loop is the sum, over all times round the loop, of the time to execute the body and the time for evaluating the condition". An example of an informal analysis where this rule is not applied is that of Dijkstra's algorithm for finding the shortest paths in a directed graph from a given vertex to every other vertex. If the adjacency matrix representation of the graph is used then the traditional informal analysis of the algorithm (see for instance /AHU82/) uses the second part of the informal rule stated in Section 1.2: "often the run-time of the loop is, neglecting constant factors, the product of the number of times the loop is executed and the maximal run-time for the body of the loop". If, on the other hand, the adjacency list representation of the graph is used, then the traditional informal analysis uses a book-keeping trick so neither of the informal rules of Section 1.2 are used. We shall in Chapter 6 discuss to what extent these two informal analyses of Dijkstra's graph algorithm can be formalised in the proof system for proving upper bounds on the run-time of programs.

In Section 6.1 we introduce the proof system for proving upper bounds on the run-time. We show that it is derived from the proof system of Chapter 3 and as an example of the use of the proof system we consider the bubble sorting algorithm. In the sections 6.2 and 6.3 we prove a negative and a positive completeness result for the proof system. Dijkstra's graph algorithm is analysed in Section 6.4 and Section 6.5 contains a discussion of how well the informal rules of Section 1.2 have been formalised in the proof system.
2 TOTAL CORRECTNESS OF WHILE PROGRAMS

As mentioned in the previous chapter we shall base the development of proof systems for run-time properties on Hoare-like proof systems for total correctness. In this chapter we shall introduce a language of while programs and we shall construct a proof system for proving total correctness of its programs. The formulas of this proof system will have post-conditions that express relationships between initial and final states - very much as in Manna and Pnueli's approach in /MaPn74/. As we argued in Section 1.3 this will be important for the development presented in the next chapters.

The language of while programs is defined in Section 2.1. It is parameterised on a data type and given a model for this we can define the semantics of the while programs rigourously. The proof system $T$ for total correctness of while programs is given in Section 2.2. It is proved to be sound and complete (in the sense of Cook) in the sections 2.3 and 2.4, respectively. Finally, we shall in Section 2.5 give some concluding remarks. Among other things, we shall compare the proof system $T$ with alternative ones presented in the literature.

2.1 THE WHILE LANGUAGE

Algorithms are usually written in a high level programming language and operate on data structures of some collection of data types. These data structures can conveniently be described using an algebraic frame-
A DATA TYPE AND ITS MODEL

The definition of our programming language will be parameterised on a data type. Syntactically, we shall specify the data type by a finite set $K$ of sorts and a $K$-sorted signature $\Sigma$. The set of sorts intuitively represents the different sorts of objects of the data type. We shall assume that $K$ always contains the element $\text{nat}$ intended to be the sort of the natural numbers.

The signature $\Sigma$ specifies the operations that may be performed on the objects. We shall distinguish between three types of operations. First, the signature may for each sort $k$ of $K$ specify certain constant symbols, that is, it gives names to certain objects of the given sort. We shall assume that $\Sigma$ always specifies the two constant symbols $0$ and $1$ of sort $\text{nat}$. Next, the signature may specify certain function symbols and for each of these it will furthermore specify an arity, that is, a pair $(k_1, \ldots, k_m, k)$ with $k_1, \ldots, k_m (m > 0)$ and $k$ being sorts from $K$. We shall assume that $\Sigma$ always specifies a function symbol $+$ of arity $(\text{nat}, \text{nat}, \text{nat})$ reflecting that $+$ takes two arguments of sort $\text{nat}$ and gives a result of sort $\text{nat}$. Finally, the signature specifies certain relation symbols and for each of these it specifies an arity being a sequence $k_1 \ldots k_m$ of sorts from $K (m > 0)$. For each sort $k$ of $K$ we shall require that $\Sigma$ specifies a relation symbol $=^k$ (often abbreviated $=$) of arity $k \cdot k$. We shall require that the signature only specifies a countable number of symbols.

Example 2.1-1. The data type of Peano Arithmetic can be specified as follows. We have just one sort, $\text{nat}$, and the $\{\text{nat}\}$-sorted signature...
specifies the following symbols:

- 0 and 1 are constant symbols of sort nat.
- + and * are function symbols and they both have arity (nat nat,nat).
- = and < are relation symbols and they both have arity nat nat.

Note, the syntax presented here for Peano Arithmetic differs slightly from the usual one in textbooks about mathematical logic. In /Sh67/, for instance, 0 is the only constant symbol and then there is a function symbol S (for successor) of arity (nat,nat).

For our purposes it is convenient to specify the implementation of the data type by giving a model for it. The model will specify the objects of the various sorts and the meaning of the various symbols of the signature. To be more formal let us consider a data type specified by a K-sorted signature Σ. A model M for this data type will to each sort k of K associate a set M_k. To each symbol of Σ it will associate an interpretation as follows:

- A constant symbol of sort k is interpreted as an element of M_k.
- A function symbol of arity (k_1,...,k_m,k) is interpreted as a total function of functionality M_{k_1} × ... × M_{k_m} → M_k.
- A relation symbol of arity k_1,...,k_m is interpreted as a relation on M_{k_1} × ... × M_{k_m}; however, the symbol = is interpreted as the identity relation on M_k for each sort k of K.

Notationally, we shall not distinguish between a symbol of the signature and its interpretation in a given model. From the context it will always be clear whether we refer to the syntactic or the semantic entity.

Note, that we here use a non-standard terminology. In a textbook on mathematical logic such as /Sh67/ M is called a structure (and a
model is something else). However, we shall later (in Chapter 3) extend $\mathcal{M}$ to specify the run-time requirements of the operations as well. Using the traditional terminology from algorithm analysis (see for instance /AHU74/) we shall call this extension of $\mathcal{M}$ for a computational model. So it seems convinient to introduce $\mathcal{M}$ under the name of a model.

Example 2.1-2. We shall now specify a model $\mathcal{N}$ for the data type of Peano Arithmetic introduced in Example 2.1-1. The set $\mathcal{N}_{\text{nat}}$ is the set of natural numbers $\{0, 1, \ldots\}$. The symbols 0 and 1 are interpreted as the natural numbers 0 and 1, respectively. The symbols $+$ and $\cdot$ are interpreted as ordinary addition and multiplication, respectively. The symbol $<$ is interpreted as the relation "less than" on the natural numbers. Using the terminology of mathematical logic (/Sh67/) we shall call $\mathcal{N}$ the standard model of Peano Arithmetic.

SYNTAX AND SEMANTICS OF THE WHILE LANGUAGE

A data type specified by a $K$-sorted signature $\Sigma$ as above defines the terms and boolean expressions of our programming language. In the following let $X$ be a $K$-sorted set of program variables. For later use we shall require that $X$ contains a countable infinite number of variables for each sort.

The terms (over $X$) can be defined by

- a variable of $X$ of sort $k$ is a term of the same sort $k$
- a constant symbol of sort $k$ is a term of the same sort $k$
- if $f$ is a function symbol of arity $(k_1 \ldots k_m, k)$ and if $e_1, \ldots, e_m$ are terms of sorts $k_1, \ldots, k_m$, respectively then $f(e_1, \ldots, e_m)$ is a term of sort $k$. 41
The boolean expressions (over \(X\)) are defined as follows:

- the symbols TRUE and FALSE are boolean expressions
- if \(p\) is a relation symbol of arity \(k_1 \ldots k_m\) and if \(e_1, \ldots, e_m\) are terms (over \(X\)) of sorts \(k_1, \ldots, k_m\), respectively, then \(p(e_1, \ldots, e_m)\) is a boolean expression
- if \(b\) and \(b'\) are boolean expressions then so are \(\neg b, b \land b', b \lor b', b \rightarrow b', b \iff b'\).

In other words, the terms and boolean expressions are, respectively, the terms and quantifier free formulas of a first order language over \(\Sigma\) (see /Sh67/).

The language of while programs (over the \(K\)-sorted signature \(\Sigma\) and the \(K\)-sorted set \(X\) of program variables) is now defined by

- if \(x\) is a variable and \(e\) is a term of the same sort as \(x\) then \(x:=e\) is a while program
- if \(c\) and \(c'\) are while programs then so are IF \(b\) THEN \(c\) ELSE \(c'\), \(c;c'\) and WHILE \(b\) DO \(c\) (if \(b\) is a boolean expression).

This definition ensures that all programs are "well-typed".

We shall often be interested in the set of free variables occurring in a program \(c\). It is denoted \(FV(c)\) and the definition in the evident one:

\[
\begin{align*}
FV(x:=e) &= FV(e) \cup \{x\} \\
FV(\text{IF } b \text{ THEN } c \text{ ELSE } c') &= FV(b) \cup FV(c) \cup FV(c') \\
FV(c;c') &= FV(c) \cup FV(c') \\
FV(\text{WHILE } b \text{ DO } c) &= FV(b) \cup FV(c)
\end{align*}
\]

where \(FV(e)\) and \(FV(b)\) are the sets of free variables of the term \(e\) and the boolean expression \(b\), respectively.
Having defined the syntax of the while language we now turn to its semantics. Several formalisms have been developed for the specification of the semantics of programming languages. We shall here use a variant of an operational approach suggested by Plotkin in /Pl81/ (and /Pl82/). The traditional operational approaches are based on some kind of abstract machine and the meaning of a program is defined in terms of the actions of the machine. The Vienna Definition Language of /BjJo78/ is an example of such an approach. It has often been argued that these approaches are too implementation oriented. Plotkin's approach is more abstract and in facts axiomatises the behaviour of the possible machine models.

In Plotkin's method the meaning of a program is defined by a transition system axiomatising the single steps of the computation. We shall use a variant of this approach and axiomatise the whole computation. More precisely, given a data type and a model for it we shall specify an "initial state - final state" relation for each of the constructs of the language.

Given a data type and a model $\mathcal{M}$ for it, we define a state as an assignment of values from the appropriate sets $\mathcal{M}_k$ to the program variables of $X$. Given a state $s$ the values of the terms and boolean expressions can be determined. If $\ e$ is a term of sort $k$ then we shall write $\ e(s)$ for the element of $\mathcal{M}_k$ being the value of $\ e$ in the state $s$. Similarly, $\ b(s)$ denotes the truth value of the boolean expression $\ b$ in the state $s$ and we shall write $\ #b(s)$ to mean that $b(s)$ is true.

The "initial state - final state" relation for a program $c$ will be written $\langle c,s \rangle \rightarrow s'$, the intuitive meaning being that if the execution of $c$ starts in the state $s$ then it will terminate and the final state
is $s'$. The relation is specified by the set $\mathcal{S}_0$ of axioms and rules given in the table below.

Semantics of while programs: $\mathcal{S}_0$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ass-}\mathcal{S}_0$</td>
<td>$\langle x:=e,s \rangle \Rightarrow s'_{x}$</td>
</tr>
<tr>
<td>$\text{IF-}\mathcal{S}_0$</td>
<td>$\frac{\text{IF } b \text{ THEN } c \text{ ELSE } c',s \Rightarrow s'}{\text{IF } \neg b(s), c',s \Rightarrow s'}$</td>
</tr>
<tr>
<td>$\text{WHILE-}\mathcal{S}_0$</td>
<td>$\frac{b(s), \langle c,s \rangle \Rightarrow s', \langle \text{WHILE } b \text{ DO } c,s' \rangle \Rightarrow s''}{\langle \text{WHILE } b \text{ DO } c,s \rangle \Rightarrow s''}$</td>
</tr>
</tbody>
</table>

The axioms and rules of $\mathcal{S}_0$ should be straightforward to understand. We use $s^v_x$ to denote the state that is as $s$ except that the value of $x$ is $v$. The formal system $\mathcal{S}_0$ can also be found in /GrMe81/.

**Example 2.1-3.** Consider the following program written in the language of while programs and using the data type of Peano Arithmetic (Example 2.1-1):

```plaintext
WHILE y<x DO (y:=y+1; fac:=y'fac)
```

Given the standard model of Peano Arithmetic (Example 2.1-2) we can deduce that
\[ \text{\textbf{Properties of While Programs}} \]

In this subsection we shall list a few properties of while programs that will be needed in later sections. We shall first introduce an equivalence relation on states. Given a finite set \( V \) of variables we define for two states \( s \) and \( s' \), \( s \equiv_V s' \) to mean that for every variable \( x \) of \( V \), \( x(s) = x(s') \), that is, \( s \) and \( s' \) are equal on \( V \).

The first property expresses that a program \( c \) does not change the values of variables that does not occur in it:

**Lemma 2.1-1.** If \( \langle c, s \rangle \rightarrow s' \) and \( V \cap \text{VFV}(c) = \emptyset \) for a finite set \( V \) of variables then \( s \equiv_V s' \). ///

The proof of this result is straightforward by induction on the length of the proof of \( \langle c, s \rangle \rightarrow s' \) in \( S_0 \).
The second lemma allows us to change the values of variables that does not occur in the program:

Lemma 2.1-2. If \( \langle c, s \rangle \rightarrow s' \) and \( s \notin \mathcal{V}_{s_0} \) for a finite set \( \mathcal{V} \) of variables satisfying \( \text{FV}(c) \subseteq \mathcal{V} \) then \( \langle c, s_0 \rangle \rightarrow s_0' \) for some state \( s_0' \) with \( s_0' \notin \mathcal{V}_{s'} \).

Also the proof of this result is straightforward by induction on the length of the proof of \( \langle c, s \rangle \rightarrow s' \) in \( \mathcal{S}_0 \).

Finally, we shall need a result stating that the while language is deterministic:

Lemma 2.1-3. If \( \langle c, s \rangle \rightarrow s' \) and \( \langle c, s \rangle \rightarrow s'' \) then \( s' = s'' \).

This result can be proved by induction on the length of the proof of \( \langle c, s \rangle \rightarrow s' \) in \( \mathcal{S}_0 \). Given the last axiom or rule applied in that proof we can deduce that the same axiom or rule must be applied in the proof of \( \langle c, s \rangle \rightarrow s'' \) since we have the same program and the same initial state. We omit the details.

### 2.2 THE PROOF SYSTEM \( \mathcal{T} \)

Having defined the syntax and semantics of the while language we now turn to the development of a total correctness proof system called \( \mathcal{T} \). The formulas of this proof system have the form \( P \langle c \rangle Q / \mathcal{V} \). Here \( \mathcal{V} \) is a finite set of program variables, intuitively, it contains the variables we are interested in. We shall therefore require that the free variables of the program \( c \) are included in \( \mathcal{V} \). The pre-condition \( P \) expresses a property of the initial state of the computation whereas the post-condition \( Q \) expresses a relation between the initial and the final state. Both \( P \) and \( Q \) are formulas of the assertion language. The
set \( V \) is introduced as a component in the formulas of the proof system in order to be able to express the post-conditions as formulas in the assertion language. We shall discuss its importance in detail later. Before giving the axioms and rules of the proof system \( \mathcal{T} \) we shall need some preliminaries.

**The Assertion Language**

In the previous section we defined the while language on top of a data type specified by a \( K \)-sorted signature \( \Sigma \) and we introduced a \( K \)-sorted set \( X \) of program variables. The assertion language will also be defined on top of this data type, however, we shall need an extended set \( X' \) of variables (including the set \( X \)). More precisely, the assertion language is the first-order language over the \( K \)-sorted signature \( \Sigma \) using the variables of \( X' \). So the formulas are defined by

- any boolean expression (over \( X' \)) is a formula
- if \( Q \) and \( Q' \) are formulas then so are \( \neg Q, Q \land Q', Q \lor Q', Q \rightarrow Q', Q \iff Q' \), \( \forall x.Q \) and \( \exists x.Q \) where \( x \) is in \( X' \).

As usually, we let \( \text{FV}(Q) \) denote the set of free variables of the formula \( Q \).

We shall now specify the set \( X' \) in more detail. In order to express the required post-conditions of the formulas \( P \prec Q/V \) we shall distinguish between the program variables and some other variables called shadow variables. For each program variable \( x \) we shall assume that \( X' \) contains a distinct shadow variable \( \bar{x} \) of the same sort as \( x \). We let \( \bar{X} \) denote the set of shadow variables corresponding to the set \( X \) (of program variables) and we shall require that \( X \cap \bar{X} = \emptyset \) and \( X' = X \cup \bar{X} \) (slightly different conditions will be imposed in Chapter 3).
The correspondence between program variables and shadow variables is used in the specification of the post-conditions. The formula $Q$ of $P<\triangleright Q/V$ has to express a relation between the initial and the final state of the computation. Syntactically, this is accomplished by using the shadow variable $\bar{x}$ to refer to the value of the variable $x$ in the initial state and let $x$ itself refer to the value of $x$ in the final state. A formula with both program and shadow variables as free variables will be called a relational formula (confer the concept of a relational predicate used by for instance /Ac82/).

A pure formula is a formula with only program variables as free variables (confer the pure predicates of /Ac82/). The pre-condition $P$ of a formula $P<\triangleright Q/V$ will be a pure formula.

Given a model $\mathcal{M}$ for the data type we defined in Section 2.1 a state as an association of values (of appropriate sorts) with the program variables, that is, the variables of the set $X$. The truth of a relational formula $Q$ in $\mathcal{M}$ shall be written as $\mathcal{F}Q(s,s')$ where $s$ and $s'$ stand for the initial and the final state, respectively. Thus $s$ and $s'$ can be viewed as the "shadow part" and the "program part", respectively, of an extended state that associates values with the variables of $X'$. For a pure formula $P$ we shall write $\mathcal{F}P(s)$ for the truth of $P$ in $\mathcal{M}$. Here $s$ will be the "program part" of an extended state. Finally, we note that we in $\mathcal{F}Q(s,s')$ and $\mathcal{F}P(s)$ omit a subscript $\mathcal{M}$ indicating that we have truth in $\mathcal{M}$. It will always be clear what model we refer to.

Example 2.2-1. Given the data type of Peano Arithmetic (Example 2.1-1) the following is an example of a formula in the proof system:

$$y=1\langle \text{WHILE }y<x\text{ DO (y:=y+1; fac:=y'fac)fac=fac'x!};/\{x,y,fac\}\rangle$$

We have here written fac=fac'x! for the relational formula expressing
that the final value of fac is the product of the initial value of fac and the factorial of x. The formula can be expressed straightforwardly within the assertion language (for example by coding the sequence $1!, 2!, \ldots, x!$ as a single natural number).

To simplify the notation used in the axioms and rules to be presented below we shall introduce some abbreviations. In the following let $X$ be a vector of the program variables in the finite set $V$ of variables and let $\overline{X}$ be the corresponding vector of shadow variables. For each pure formula $P$ we define the relational formula $\overline{P}$ to be $P^{\overline{X}}_X$. If $\text{FV}(P) \subseteq V$ then $\overline{P}$ only contains shadow variables as free variables and for every pair $(s, s')$ of states we have

$$\forall P(s) \text{ if and only if } \exists \overline{P}(s, s').$$

Similarly, for each term $e$ of the expression language we shall write $\overline{e}$ for $e^{\overline{X}}_X$.

For two relational formulas $Q$ and $Q'$ we write $Q \cdot Q'$ as an abbreviation for the formula $\exists X'. Q^{X'}_X \land Q'^{\overline{X'}}_{\overline{X}}$ where $X'$ is a vector of distinct new variables of the same length as $X$ (and $\overline{X}$) and of the appropriate sorts. Assume now that $\text{FV}(Q) \cup \text{FV}(Q') \subseteq \overline{V}$ where $\overline{V}$ is the set of shadow variables corresponding to the program variables of $V$. Then for every pair $(s, s')$ of states we have

$$\forall Q \cdot Q'(s, s') \text{ if and only if } \forall Q(s, s'') \land \exists Q'(s'', s') \text{ for some } s''.$$

Finally, we shall use $I_V$ as an abbreviation for the formula

$$\ldots \land x = \overline{x} \land \ldots$$

for all the (finitely many) variables $x$ of $V$. For every pair $(s, s')$ of states we have

$$\forall I_V(s, s') \text{ if and only if } s \in V s'.$$
The Proof System

As mentioned, the formulas of the proof system $\mathcal{J}$ are going to have the form $P(c)Q/V$ where $c$ is a while program, $P$ a pure formula, $Q$ a relational formula, and $V$ a finite set of program variables. We shall impose a well-formedness condition on these formulas and, intuitively, it will ensure that the program variables we are interested in are included in the set $V$. Formally, $P(c)Q/V$ is a well-formed formula of $\mathcal{J}$ if

- $\text{FV}(c) \subseteq V$
- $\text{FV}(P) \subseteq V$ and $\text{FV}(Q) \subseteq V$

The validity of a well-formed formula (written $\mathcal{J}P(c)Q/V$) is now defined as follows for a given model $\mathcal{M}$ for the data type

for every state $s$ satisfying $\mathcal{M}P(s)$ there is a state $s'$ such that $<c,s> \rightarrow s'$ and $\mathcal{M}Q(s,s')$

Note that this definition captures the idea that if $P$ holds on the initial state then the program will terminate and $Q$ will hold for the initial and the final state.

The axioms and rules of the proof system $\mathcal{J}$ are listed in the table below.

We shall write $\mathcal{J} \vdash P(c)Q/V$ if the formula $P(c)Q/V$ is provable from the axioms and rules of $\mathcal{J}$, however, an axiom or rule can only be applied if the formulas of the assertion language appearing in the hypothesis of the rule are true and if the conclusion of the axiom or rule is a well-formed formula of $\mathcal{J}$.
The proof system $\mathcal{T}$

/ass-$\mathcal{T}$/ \[ P(x:=e)V \vdash x=e/V \]

/IF-$\mathcal{T}$/ \[ P(a\beta c_1)Q/V, \quad P(a\beta c_2)Q/V \]
\[ P(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2)Q/V \]

/;$\mathcal{T}$/ \[ P(c_1)P'\land_1 Q/V, \quad P(c_2)P_2 Q/V \]
\[ P(c_1;c_2)Q_1\land_2 Q/V \]

/WHILE-$\mathcal{T}$/ \[ P(z+1)\lambda b c P(z)A'Q'/V, P(0)\rightarrow \beta, \]
\[ P(z)\lambda b a Q/V, \quad Q'\rightarrow Q \]
\[ \exists z. P(z)\langle \text{WHILE } b \text{ DO } c\rangle Q/V \]
where $z$ is a variable of sort nat satisfying $z\notin V$

/cons-$\mathcal{T}$/ \[ P\rightarrow P', \quad P'\langle c\rangle Q'/V, \quad Q'\rightarrow Q \]
\[ P\langle c\rangle Q/V \]

/inv-$\mathcal{T}$/ \[ P\langle c\rangle Q/V. \quad P\langle c\rangle P_1 Q/V \]

We shall now give an informal explanation of the axioms and rules of $\mathcal{T}$. In /ass-$\mathcal{T}$/ the post-condition expresses that the values of all variables except $x$ are unchanged and $x$ has got the value of $e$ in the initial state. In /IF-$\mathcal{T}$/ we assume that the two branches give rise to the same post-conditions (the rules /cons-$\mathcal{T}$/ and /inv-$\mathcal{T}$/ can be used to ensure that this is the case).

In /;$\mathcal{T}$/ we assume that the post-condition of the first component $c_1$ has been split into a pure part, $P'$, and a relational part, $Q_1$. As we shall see in Section 2.4 this will (under reasonable assumptions) always be possible. The second component $c_2$ gives rise to another
post-condition \( Q_2 \), and the composition of \( Q_1 \) and \( Q_2 \) describes the effect of executing the composite program. This is illustrated by the following figure:

In /WHILE-T/ the idea is that the value of \( z \) bounds the number of unfoldings of the while loop. The post-condition for the loop body \( c \) is split into two parts, a pure part expressing that now less unfoldings are required and a relational part \( Q' \) describing the effect of executing the body once. The relational formula \( Q \) expresses the effect of executing the body a number of times. The following figure illustrates the connection between \( Q' \) and \( Q \):

The rules /cons-T/ and /inv-T/ should be straightforward to understand.
As an example of the use of the proof system $\mathcal{J}$ let us sketch a proof of the formula

$$y_1 < \text{WHILE } y < x \text{ DO } (y := y + 1; \text{ fac} := y \cdot \text{ fac}) \Rightarrow \text{ fac} = \text{ fac} \cdot x! \leftrightarrow \{x, y, \text{ fac}\}$$

considered in Example 2.2-1. We shall use the data type of Peano Arithmetic (Example 2.1-1) and its standard model (Example 2.1-2). In the following let $V$ be an abbreviation for the set $\{x, y, \text{ fac}\}$.

The proof will be presented in a bottom-up manner. So we shall first construct a proof for the body of the loop. The invariant $P(z)$ for the loop is chosen to be the formula

$$(x = 0 \land z = 0 \land y = 1) \lor (x > 0 \land x = y + z)$$

From the axiom $/\text{ass-}\mathcal{J}/$ we then get a proof of

$$P(z+1) \land y < x \land (y := y + 1) \Rightarrow I_{\{x, \text{ fac}, z\}} ^{y = y + 1} /\lor \{z\}$$

Using $/\text{inv-}\mathcal{J}/$ we then get

$$P(z+1) \land y < x \land (y := y + 1) \Rightarrow P(z+1) \land I_{\{x, \text{ fac}, z\}} ^{y = y + 1} /\lor \{z\}$$

It can easily be verified that

$$P(z+1) \land y < x \land (y := y + 1) \Rightarrow P(z) \land I_{\{x, \text{ fac}\}} ^{y = y + 1}$$

so using $/\text{cons-}\mathcal{J}/$ we obtain a proof of

$$P(z+1) \land y < x \land (y := y + 1) \Rightarrow P(z) \land I_{\{x, \text{ fac}\}} ^{y = y + 1} /\lor \{z\}$$

Similarly, we can prove

$$P(z) \land \text{ fac} := y \cdot \text{ fac} \Rightarrow P(z) \land I_{\{x, y\}} ^{\text{ fac} = y \cdot \text{ fac}} /\lor \{z\}$$

These two proofs can now be put together using the rule $/\text{;}\mathcal{J}/$ and we get a proof of
It is straightforward to verify that

\[(I_x, fac) \land y = y+1) \land (P(z) \land I_{x,y}) \land fac = \overline{y'fac}) \rightarrow
\]
\[P(z) \land fac = \overline{y'fac} \land x \land ay = ay + 1\]

so using /cons-\(T\)/ we get the following proof for the body of the loop:

\[P(z+1) \land x \land y = y+1; \land fac = \overline{y'fac} > P(z) \land fac = \overline{y'fac} \land x \land ay = ay + 1 / Vu[z]\]

In order to get a proof for the while loop we shall use that

\[\forall P(0) \rightarrow \neg (y < x)\]
\[\forall P(z) \land \neg (y < x) \land I_v \rightarrow fac = \overline{fac'x!/y!}\]

and

\[\forall (fac = \overline{y'fac} \land x \land ay = ay + 1) \land (fac = \overline{fac'x!/y!}) \rightarrow fac = \overline{fac'x!/y!}\]

We can then apply the rule /WHILE-\(T\)/ and get a proof of

\[\exists z. P(z) < \text{WHILE } y < x \text{ DO } (y = y+1; \land fac = \overline{y'fac}) > fac = \overline{fac'x!/y!} / V\]

In order to get the required proof we shall first apply /cons-\(T\)/ with

\[\forall y = 1 \rightarrow \exists z. P(z)\]

then /inv-\(T\)/ and finally /cons-\(T\)/ with

\[\forall y = 1 \land fac = \overline{fac'x!/y!} \rightarrow fac = \overline{fac'x!}\]

This completes the proof.

2.3 THE SOUNDNESS THEOREM FOR T

We now turn to a discussion of the theoretical properties of soundness and completeness of the proof system T. The soundness property says, intuitively, that anything that can be proved in the proof
system does indeed hold, that is, given a data type and a model for it then for every well-formed formula $P \langle c \rangle Q / V$

$$T \vdash P \langle c \rangle Q / V \text{ implies } \forall P \langle c \rangle Q / V$$

This is the kind of soundness result that can be obtained for a proof system for partial correctness as that considered in Section 1.1. However, this does not hold for a total correctness proof system as $T$. To see this consider the data type of Peano Arithmetic introduced in Example 2.1-1. This data type has a (non-standard) model, say $M$, with the same theorems as its standard model which we gave in Example 2.1-2. Consider now the formula

$$\text{TRUE} \langle \text{WHILE } y < x \text{ DO } y := y + 1 \rangle \text{TRUE} / \{x, y\}$$

It is straightforward to construct a proof of this formula in $T$ using only formulas of the assertion language that are true in $M$ (and thereby the standard model). However, the formula is not valid. The set $M_{\text{nat}}$ will contain a non-standard value, say $v$ (that is, $v$ is not a successor of 0). Let $s$ be a state with $x(s) = v$ and $y(s) = 0$. Started from this state the program $\text{WHILE } y < x \text{ DO } y := y + 1$ will never terminate so the formula cannot be valid. This proves that the proof system $T$ cannot be sound in the sense that for any data type and any model for it provability of a formula $P \langle c \rangle Q / V$ implies validity of the formula.

**THE SOUNDNESS THEOREM AND ITS PROOF**

The usual approach is therefore to use a weaker notion of soundness. Given a data type and a model $M$ for it we shall call $M$ a **numerical model** if

- $M_{\text{nat}}$ is the set $\mathbb{N}$ of natural numbers
- the symbols 0 and 1 are interpreted as the natural numbers 0 and 1, respectively, and the symbol + is interpreted as ordinary addition.

The standard model of Peano Arithmetic given in Example 2.1-2 is an example of a numerical model.

We have the following result:

The Soundness Theorem for $\mathcal{T}$

Given a data type and a numerical model for it, then for every well-formed formula $P(c)Q/V$ of $\mathcal{T}$

\[ \mathcal{T}P(c)Q/V \text{ implies } \mathcal{F}P(c)Q/V \]

To prove this result it is sufficient to prove that the axiom of $\mathcal{T}$ is valid and that the rules of $\mathcal{T}$ preserve validity. Then the theorem follows by induction on the length of proofs in $\mathcal{T}$.

Case /ass-$\mathcal{I}$/: We shall prove that

\[ \mathcal{F}P(x:e)I_{V-\{x\}}^xA_{x:e/V} \]

holds. So assume that $\mathcal{F}P(s)$ holds for some state $s$. From /ass-$\mathcal{I}$/ we have

\[ \langle x:e,s \rangle \rightarrow s^e_x(s) \]

and from the definition of $I_{V-\{x\}}$ we easily get that $\mathcal{F}I_{V-\{x\}}(s,s^e_x(s))$ holds. It is also easy to see that $\mathcal{E}x:e(s,s^e_x(s))$ holds and the required validity follows.

Case /IF-$\mathcal{I}$/: We shall prove that the rule preserves validity so assume that

(1) $\mathcal{F}Pab(c)Q/V$
and $\Diamond P \land b <c_2> Q/V$

We have to prove that $\Diamond P <\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2> Q/V$

so assume that $\Diamond P(s)$ holds for some state $s$. In the case where $Pb(s)$ holds we get from (1) that

$<c_1,s> \rightarrow s'$ and $Q(s,s')$

for some state $s'$. Using $\text{IF- }$ we then get $<\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2,s> \rightarrow s'$

and the result follows. The case where $\neg b(s)$ holds is similar.  ///

Case $i: \neg P$: We assume that

(1) $\Diamond P <c_1> P' \land Q_1/V$

and

(2) $\Diamond P' <c_2> Q_2/V$

and we have to prove that $\Diamond P <c_1;c_2> Q_1,Q_2/V$.

So assume that $\Diamond P(s)$ holds for some state $s$. From (1) we get that $<c_1,s> \rightarrow s'$ and $P' \land Q_1(s,s')$

for some $s'$. Then $\Diamond P'(s')$ and the assumption (2) gives $<c_2,s'> \rightarrow s''$ and $Q_2(s',s'')$

for some $s''$. But then $\text{IF- }$ gives $<c_1;c_2,s> \rightarrow s''$

and from $\Diamond Q_1(s,s')$ and $\Diamond Q_2(s',s'')$ we get $\Diamond Q_1,Q_2(s,s')$. This proves the required validity.  ///
Case /WHILE-\exists/: We shall prove that the rule preserves validity so assume that

(1) \( P(z+1) \land b \langle c \rangle P(z) \supset Q'/VU\{z\} \)

(2) \( P(0) \supset \neg b \)

(3) \( P(z) \land b \supset Q \)

and

(4) \( Q' \supset Q \)

where \( z \) is a variable of sort \( \text{nat} \) satisfying \( z \notin V \). We have to prove

\( \exists z. P(z) \langle \text{WHILE} \ b \ \text{DO} \ c \rangle Q/V \)

so assume that \( \exists z. P(z)(s) \) for some state \( s \). Since the model is numerical this means that for some natural number \( n, P(z)(s^n) \). By induction on \( n \) we shall now prove that for some state \( s' \)

(\( ) \langle \text{WHILE} \ b \ \text{DO} \ c, s \rangle \rightarrow s' \) and \( Q(s, s') \).

If \( n=0 \) then (1) \( P(0)(s) \) and (2) gives \( \neg b(s) \). Therefore /WHILE-\( 0 \)/ gives

\( \langle \text{WHILE} \ b \ \text{DO} \ c, s \rangle \rightarrow s. \)

Since \( FV(b) \subseteq V \) we have \( P(z) \supset b \supset Q(s^n, s^n) \) and from (3) we get \( Q(s^n, s^n) \).

Using that the conclusion of the rule /WHILE-\( 0 \)/ is well-formed we get \( FV(Q) \subseteq V \) and thereby \( Q(s, s) \). This proves (\( ) \) for \( n=0 \).

Assume now that (\( ) \) holds for \( n=n' \) and let us prove it for \( n=n'+1 \). So assume that \( P(z)(s^{n'+1}) \). If \( \neg b(s) \) holds we proceed as above in the case \( n=0 \) so assume that \( b(s) \) holds. Then \( P(z+1) \land \neg b(s^n) \) holds (using that \( FV(b) \subseteq V \)) so from (1) we get that for some state \( s'' \)

\( \langle c, s^n, s^n \rangle \rightarrow s'' \) and \( P(z) \supset Q'(s^n, s^n) \).

From Lemma 2.1-1 we get \( z(s'')=n' \) since \( z \notin FV(c) \) (the well-formedness
of the conclusion of the rule /WHILE-\text{T}/. Thus $P(z)(s'^n_z)$ so the induction hypothesis gives

$$\langle \text{WHILE } b \text{ DO } c, s'' \rangle \rightarrow s' \text{ and } \exists Q(s'', s')$$

for some state $s'$. Then /WHILE-\text{\_T}/ gives

$$\langle \text{WHILE } b \text{ DO } c, s^m_z \rangle \rightarrow s'_0$$

for some state $s'_0$ with $s'_0 \models s'$. From $\exists Q'(s'^n_z, s'')$ and $\exists Q(s'', s')$ we get $\exists Q'(s'^n_z, s')$ and thus using (4) $\exists Q(s'^n_z, s')$. Since $\exists Q(s', s'_0)$ we get $\exists Q(s, s'_0)$. This completes the proof of (§).

Case /cons-\text{T}/: Straightforward and therefore omitted.

Case /inv-\text{T}/: Straightforward and therefore omitted.

This completes the proof of The Soundness Theorem for \text{T}.

2.4 THE COMPLETENESS THEOREM FOR \text{T}

The completeness property is the converse of that of soundness. As mentioned in Chapter 1 Wand has proved that Hoare's proof system for partial correctness is not complete in that sense /Wa78/. Cook suggested therefore to introduce an expressiveness condition on the assertion language relative to its model and the programming language /Co78/. Under the assumption that the expressiveness condition is fulfilled it can then be proved that Hoare's proof system is complete.

We shall use a similar approach here and impose an expressiveness condition that will ensure that we get a completeness result for the proof system \text{T}. Harel seems to be the first to prove completeness of
a proof system for total correctness /Ha79/. His approach is slightly
different from ours and we shall therefore close this section with a
review of it.

**THE EXPRESSIVENESS CONDITION**

The expressiveness condition will ensure that the graphs of the
programs are expressible as formulas of the assertion language. More
precisely, given a data type and a numerical model for it we shall
say that the expressiveness condition for \( T \) is fulfilled if

for every while program \( c \) there is a relational formula \( G[c] \) with
\( \text{FV}(G[c]) \subseteq \text{FV}(c) \cup \text{FV}(c) \) and satisfying that for every pair \( (s,s') \) of
states

\[ F[c](s,s') \]

if and only if

\[ \langle c,s \rangle \rightarrow s'' \text{ for some state } s'' \text{ with } s'' = \text{FV}(c)s'. \]

An example of a data type and a model satisfying this condition is
Peano Arithmetic (Example 2.1-1) and its standard model (Example
2.1-2). This follows from results proved later in this section (in
fact from Lemma 2.4-1).

The definition above is equivalent to the usual one that requires
that the strongest post-conditions are definable, that is, given a
(pure) formula \( P \) and a while program \( c \) there is a (pure) formula
\( \text{SP}(P,c) \) such that for every state \( s' \), \( \text{SP}(P,c)(s') \) if and only if
\( \langle c,s \rangle \rightarrow s' \) and \( P(s) \) hold for some state \( s \). In /0180/, Olderog has
proved that this definition is equivalent to one requiring the weakest
pre-conditions to be definable. Some alternative (and weaker)
expressiveness conditions have been suggested by Sieber in /Si83/.
THE COMPLETENESS THEOREM AND ITS PROOF

Using this notion of expressiveness we have the following result.

The Completeness Theorem for $\mathcal{T}$

Given a data type and a numerical model for it, if the expressiveness condition for $\mathcal{T}$ is fulfilled then for every well-formed formula $P(c)Q/V$:

$$\mathcal{T} - P(c)Q/V.$$  

The proof of this result is by structural induction on the program $c$.

Case $x:=e$: Assume that

1. $\mathcal{T} - P(x:=e)Q/V$

holds and we shall construct a proof of the formula in $\mathcal{T}$. From $\text{/ass-}\mathcal{T}/$ we get a proof of

$$P(x:=e)I_{\mathcal{T}-\{x\}}^{\lambda x=e/V}$$

so using $\text{/inv-}\mathcal{T}/$ we get

$$P(x:=e)P_{\mathcal{T}}I_{\mathcal{T}-\{x\}}^{\lambda x=e/V}.$$ 

Below we shall prove that

2. $\mathcal{T} - P_{\mathcal{T}}I_{\mathcal{T}-\{x\}}^{\lambda x=e} \rightarrow Q$

so using $\text{/cons-}\mathcal{T}/$ we get the required proof.

To prove (2) assume that $\mathcal{T} - P_{\mathcal{T}}I_{\mathcal{T}-\{x\}}^{\lambda x=e(s,s')}$ for some pair $(s,s')$ of states. Then $P(s)$ holds so from (1) we get

$$(x:=e,s) \rightarrow s''$$

and $PQ(s,s'')$

for some state $s''$. From $\text{/ass-}\mathcal{T}/$ and Lemma 2.1-3 we get $s'' = s^e_x$ and from $\mathcal{T} - P_{\mathcal{T}}I_{\mathcal{T}-\{x\}}^{\lambda x=e(s,s')}$ we have $s' = s^e_x(s)$. Since $FV(Q)\mathcal{E}V\mathcal{W}^V (P(x:=e)Q/V$
is well-formed) we get $\mathcal{Q}(s,s')$. ///

**Case IF $b$ THEN $c_1$ ELSE $c_2$:** Assume now that

(1) $\mathcal{P}(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) \mathcal{Q}/\mathcal{V}$

and we shall construct a proof of the formula in $\mathcal{T}$. Using (1) it is straightforward to prove the validity of the two formulas

$\mathcal{P}a \mathcal{b}(c_1 \mathcal{Q}/\mathcal{V})$

and

$\mathcal{P}a \mathcal{b}(c_2 \mathcal{Q}/\mathcal{V})$

and the induction hypothesis gives a proof of both of the formulas in $\mathcal{T}$. The rule /IF-$\mathcal{T}$/ then gives the required proof. ///

**Case $c_1;c_2$:** We shall now assume that

(1) $\mathcal{P}(c_1;c_2) \mathcal{Q}/\mathcal{V}$

holds and we shall construct a proof of the formula in $\mathcal{T}$. In the following let $Q_1$, $Q_2$ and $P'$ denote the formulas $\mathcal{G}[c_1 \mathcal{N}_{V-FV(c_1)}$, $\mathcal{G}[c_2 \mathcal{N}_{V-FV(c_2)}$ and $\exists X'. \mathcal{G}[c_2 X' X]$ respectively (here $X$ is a vector of the variables of $FV(c_2)$, $\overline{X}$ is the corresponding vector of shadow variables and $X'$ is a vector of distinct new variables of appropriate sorts and of the same length as $X$). The expressiveness assumption gives that these three formulas exist and furthermore that for every state $s$

($) $\mathcal{P}'(s)$ if and only if $<c_2,s> \rightarrow s'$ for some $s'$

and for every pair $(s,s')$ of states (and for $i=1$ and $i=2$)

($\$) $\mathcal{Q}_i(s,s')$ if and only if $<c_i,s> \rightarrow s''$ for some $s''$ with $s'' \in s'$.

Below we shall prove that the formulas
and

\[ P'(c_2 \rightarrow Q_2) /V \]

are valid and the induction hypothesis then gives that they are provable in \( \mathcal{I} \). Using \( /;\mathcal{I} / \) we then get a proof of

\[ P(c_1; c_2 \rightarrow Q_1 \land Q_2) /V. \]

Below we prove that

\[ P(\mathcal{A}(Q_1 \land Q_2)) /PQ \]

so using first \( /\text{inv-} \mathcal{I} / \) and then \( /\text{cons-} \mathcal{I} / \) with (4) we get the required proof.

To prove that the formula (2) is valid assume that \( \mathcal{P}(s) \) holds for some state \( s \). Then (1) gives that for some \( s'' \)

\[ \langle c_1, c_2, s'' \rangle \rightarrow s'' \text{ and } \mathcal{P}(s, s'') \]

and according to the semantic rule \( /;\mathcal{I} / \) this means that for some state \( s' \)

\[ \langle c_1, s \rangle \rightarrow s' \text{ and } \langle c_2, s \rangle \rightarrow s'' \]

From (5) and (6) we now get that \( \mathcal{P}(\mathcal{A}Q_1(s, s')) \) and this proves the validity of (2).

It is straightforward to prove that the formula (3) is valid using (5) and (6).

To prove (4) assume that \( \mathcal{P}(\mathcal{A}Q_1 \land Q_2)(s, s'') \) for some pair \( (s, s'') \) of states. Then \( \mathcal{P}(s) \) and from (1) we get

\[ \langle c_1, c_2, s \rangle \rightarrow s'' \text{ and } \mathcal{P}(s, s'') \]

for some state \( s'' \). From \( \mathcal{P}(\mathcal{A}Q_1 \land Q_2)(s, s'') \) we get that for some \( s' \) \( \mathcal{P}(s, s') \)
and $\forall Q_2(s',s'')$. According to (§§) this means that

$$\langle c_1, s_0 \rangle \rightarrow s_0' \text{ for some } s_0' \text{ with } s_0' \models s'$$

and

$$\langle c_2, s'' \rangle \rightarrow s_2'' \text{ for some } s_2'' \text{ with } s_2'' \models s'' \text{.}$$

Since $s_0' \models s'$ and $FV(c_2) \subseteq V$ we get from Lemma 2.1-2

$$\langle c_2, s_0' \rangle \rightarrow s_2'' \text{ for some } s_2'' \text{ with } s_2'' \models s'' \text{.}$$

Now $\not\vdash_{0}$ gives that

$$\langle c_1; c_2, s \rangle \rightarrow s_2'' \text{.}$$

and from Lemma 2.1-3 we get $s_2'' = s''$. Since $FV(Q) \subseteq V$, $s'' \models s'$ and

$\forall Q(s, s'_0)$ we get $\forall Q(s, s')$ as required. //

Case WHILE b DO C: Assume now that

(1) $\forall P \langle \text{WHILE } b \text{ DO } c \rangle \not\vdash_{Q/V}$

and we shall construct a proof of the formula in $\not\vdash$. We shall now define $P'(z)$ to be a pure formula expressing that the body of the loop is executed at most $z$ times - here $z$ is a new variable of sort nat. So define $c'$ to be the program

$$\text{IF } z=0 \text{ THEN loop ELSE } (c; z:=z-1)$$

where loop is a program that never terminates and $z:=z-1$ is a program that decrements the value of $z$ by one; for instance

$$\text{loop } \equiv \text{ WHILE TRUE DO } z:=z$$

and

$$z:=z-1 \equiv z'':0; \text{ WHILE } z'+1 \leq z \text{ DO } z':=z'+1; \ z:=z'$$

where $z'$ has sort nat and $z' \not\vdash_{V \cup \{z\}}$. It can then be proved that for every pair $(s, s')$ of states
\[
\begin{aligned}
\langle c', s \rangle \rightarrow s'
\end{aligned}
\]

if and only if

\[
\begin{aligned}
\langle c, s \rangle \rightarrow s'' \text{ for some } s'' \text{ with } s'' \equiv_{V} s', \ z(s) > 0 \text{ and } z(s') = z(s) - 1.
\end{aligned}
\]

Formally, we now define \( P'(z) \) to be the pure formula

\[
\exists z'. \exists x'. \forall (\text{WHILE } b \text{ DO } c') x' = x \text{ if and only if } \langle \text{WHILE } b \text{ DO } c', s \rangle \rightarrow s' \text{ for some state } s'.
\]

In the proof below we shall also use the following property

\[
\langle \text{WHILE } b \text{ DO } c', s \rangle \rightarrow s' \text{ implies that for some natural number } n
\]

\[
\langle \text{WHILE } b \text{ DO } c', s'' \rangle \rightarrow s'' \text{ for some } s''.
\]

Furthermore, we define the two relational formulas \( Q' \) and \( Q'' \) by

\[
Q' \equiv \mathcal{G}[\text{WHILE } b \text{ DO } c]_{\mathcal{A}_V - \text{FV} (\text{WHILE } b \text{ DO } c)}
\]

and

\[
Q'' \equiv \mathcal{E}_g[\mathcal{G}[c]_{\mathcal{A}_V - \text{FV} (c)}].
\]

Using the expressiveness assumption this means that for every pair \( (s, s') \) of states

\[
\mathcal{F}Q'(s, s')
\]

if and only if

\[
\langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow s'' \text{ for some } s'' \text{ with } s'' \equiv_{V} s''
\]

and
Below we shall prove that

(2) \( \forall P'(z+1) \land b \langle c \mathbin{\supset} P'(z) \mathbin{\supset} Q'' \mathbin{\supset} \forall \exists z \exists \rangle \),

(3) \( \forall P'(0) \rightarrow \neg b \),

(4) \( \forall P'(z) \land \neg b \land \forall \exists \rightarrow Q' \)

and

(5) \( \forall Q'' \rightarrow Q' \).

Since (2) holds we can apply the induction hypothesis and get a proof in \( \mathcal{J} \) of that formula. So we can apply the rule /WHILE-\( \mathcal{J} \)/ and get a proof of

\[ \exists z. P'(z) \langle \text{WHILE } b \text{ DO } c \rangle Q'/V. \]

Below we prove that

(6) \( \exists P \rightarrow \exists z. P'(z) \)

and

(7) \( \exists P \land \exists Q' \rightarrow Q \)

so using first /cons-\( \mathcal{J} \)/ with (6), then /inv-\( \mathcal{J} \)/ and finally /cons-\( \mathcal{J} \)/ with (7) we get a proof in \( \mathcal{J} \) of the required formula.

To prove (2) assume that \( \forall P'(z+1) \land b(s) \) holds for some state \( s \). Then \( \forall P'(z) (z(s) + 1) \) and from (6) and (7) we get

\[ \langle \text{WHILE } b \text{ DO } c', s_z(z(s) + 1) \rangle \rightarrow s'' \]

for some state \( s'' \). Since \( b(s) \) we get from /WHILE-\( \mathcal{J} \)/

(8) \( \langle c', s_z(z(s) + 1) \rangle \rightarrow s' \) and \( \langle \text{WHILE } b \text{ DO } c', s' \rangle \rightarrow s'' \)

for some state \( s' \). From (8) we then get...
where $s_0' \leq s'$ and $z(s) = z(s')$. Since $s_0^{z(s)+1} \leq s$ we get from Lemma 2.1-2

(9) $\langle c, s \rangle \rightarrow s'_1$

where $s_1' \leq s_1$. From (8) we get $P''(s, s_1')$. From (8) and (9) we have $P'(z)(s')$. We have $z(s) = z(s_1')$ from Lemma 2.1-1 and (9) so we get $s' \leq V u z \leq s_1'$. Since $FV(P'(z)) \subseteq V u \{ z \}$ we therefore get $P'(z)(s_1')$. This proves (2).

To prove (3) assume that $P'(0)(s)$ and $b(s)$ for some state $s$. Then (§§) gives

$\langle \text{WHILE } b \text{ DO } c', \bar{s}_2 \rangle \rightarrow s''$

for some $s''$ and since $b(s)$ holds $\text{WHILE-} _0/ \text{ gives}$

$\langle c', \bar{s}_2 \rangle \rightarrow s' \text{ and } \langle \text{WHILE } b \text{ DO } c', s' \rangle \rightarrow s''$

for some $s'$. From (§) we then get $z(s_2') \geq 0$ but this is a contradiction. This proves (3).

To prove (4) assume that $P'(z) \forall b A I (s, s')$ for some pair $(s, s')$ of states. From $\text{WHILE-} _0/ \text{ we get}$

$\langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow s$

and (§) gives $Q'(s, s)$. Since $FV(Q') \subseteq V u \bar{v}$ and $s \leq s'$ we get $Q'(s, s')$ and this proves (4).

To prove (5) assume that $Q''(Q'(s, s'))$ for some pair $(s, s'')$ of states, that is, for some state $s'$ we have $Q''(s, s')$ and $Q'(s', s')$. From (§§) and (§) we now get

$b(s), \langle c, s \rangle \rightarrow s_0$ for some $s_0'$ satisfying $s_0' \leq s'$

and

$\langle \text{WHILE } b \text{ DO } c, s' \rangle \rightarrow s_0''$ for some $s_0''$ satisfying $s_0'' \leq s''$. 

67
Lemma 2.1-2 now gives

\[ \langle \text{WHILE } b \text{ DO } c; s_0' \rangle \rightarrow s_1' \]

for some \( s_1' \) satisfying \( s_1' \models \psi s_0' \) since \( s_0' \models \psi s' \). From \( /\text{WHILE}-\xi_0/ \) we get

\[ \langle \text{WHILE } b \text{ DO } c; s \rangle \rightarrow s_1' \]

and thus \( \xi Q'(s, s_1') \) follows from (E). We have \( s_1' \models \psi s_1' \) and \( \text{FV}(Q') \Vdash \xi \psi \xi \)

so \( \xi Q'(s, s_1') \) follows. This proves (5).

To prove (6) assume that \( \xi P(s) \) holds for some state \( s \). Then (1) gives

\[ \langle \text{WHILE } b \text{ DO } c; s \rangle \rightarrow s' \text{ and } \xi Q(s, s') \]

for some state \( s' \). From (333) we now get that for some natural number \( n \)

\[ \langle \text{WHILE } b \text{ DO } c'; s_n \rangle \rightarrow s_1' \]

for some state \( s_1' \) and thus (33) gives \( \xi P'(z)(s_n) \). Then \( \exists z. P'(z)(s) \)

and this proves (6).

To prove (7) assume that \( \xi \exists \xi Q'(s, s'') \) for some pair \( (s, s'') \) of states. Since \( \xi P(s) \) holds we get from (1) that for some state \( s' \)

\[ \langle \text{WHILE } b \text{ DO } c; s \rangle \rightarrow s' \text{ and } \xi Q(s, s') \]

Since \( \xi Q'(s, s'') \) holds we get from (E) that

\[ \langle \text{WHILE } b \text{ DO } c; s \rangle \rightarrow s_1'' \]

for some \( s_1'' \) with \( s_{1''} \models \psi s'' \). From Lemma 2.1-3 we get that \( s'=s_0'' \). From

\( \xi Q(s, s'), s' \models \psi s'' \) and \( \text{FV}(Q) \Vdash \xi \psi \xi \)

we get \( \xi Q(s, s'') \) as required. This proves (7).

This completes the proof of The Completeness Theorem for \( \xi \).  

68
ARITHMETICAL MODELS

Harel seems to be the first to prove completeness of a proof system for total correctness /Ha79/ and because his approach is slightly different from ours we shall review it here. Furthermore, it gives a sufficient condition ensuring that the expressiveness condition for $\mathcal{F}$ is fulfilled by a data type and its model.

A data type containing Peano Arithmetic (Example 2.1-1) is a data type with $'$ as a function symbol of arity (nat nat,nat) and $<$ as a relation symbol of arity nat nat. A model $\mathcal{M}$ for such a data type is called an arithmetical model if the following two conditions are fulfilled:

- $\mathcal{M}$ is a numerical model where $'$ is interpreted as ordinary multiplication of natural numbers and $<$ is interpreted as the relation "less than" on the natural numbers
- for every sort $k$ there is another sort $\hat{k}$ and a formula $\theta_{k,\hat{k}}(x,i,y)$ ($x$ has sort $k$, $i$ sort nat and $y$ sort $\hat{k}$) such that for every finite sequence $v_i, \ldots, v_n$ of elements of $\mathcal{M}_k$ there is an element $w$ of $\mathcal{M}_{\hat{k}}$ such that for every state $s$, $\theta_{k,\hat{k}}(x,i,y)(s)$ and $y(s)=w$ hold if and only if $0<i(s)<n$ and $x(s)=v_i(s)$.

The last condition ensures that a finite sequence of elements of one sort ($k$) can be encoded as a single element of another sort ($\hat{k}$).

Trivially, the data type of Peano Arithmetic (Example 2.1-1) is a data type containing Peano Arithmetic and its standard model (Example 2.1-2) is an arithmetical model (the encoding of finite sequences of natural numbers as a single natural number can for instance be accomplished by Gödel's beta function /Sh67/).
Following Harel /Ha79/ we now define arithmetical completeness of the proof system $\mathcal{T}$ to mean that given a data type containing Peano Arithmetic and given an arithmetical model for it then for every well-formed formula $P(c)Q/V$, $\mathcal{T}$ proves $P(c)Q/V$. On the other hand, if provability of the formula $P(c)Q/V$ in $\mathcal{T}$ implies that it is valid then $\mathcal{T}$ is said to be arithmetical sound. We can prove that

Corollary: The proof system $\mathcal{T}$ is arithmetical sound and complete.///

Since any arithmetical model is numerical it follows directly from The Soundness Theorem for $\mathcal{T}$ that $\mathcal{T}$ is arithmetical sound. To prove that $\mathcal{T}$ is arithmetical complete it is sufficient to prove the lemma below; The Completeness Theorem for $\mathcal{T}$ then gives the required result.

Lemma 2.4-1: The expressiveness condition for $\mathcal{T}$ is fulfilled for every data type containing Peano Arithmetic and every arithmetical model for it.

Proof: Given a program $c$ we shall construct a relational formula $G[c]$ satisfying that for every pair $(s,s')$ of states

$$G[c](s,s')$$

if and only if

$$\langle c,s \rangle \rightarrow s''$$

for some state $s''$ with $s'' \subseteq_{FV(c)} s'$. Furthermore, $FV(G[c]) \subseteq FV(c) \cup FV(c)$ has to hold. We shall define $G[c]$ by structural induction on the program $c$. For the assignment, the conditional and the composite statements we have

$$G[x:=e] = I_{FV(e)} - \{x\} \wedge \bar{e},$$

$$G[\text{IF } b \text{ THEN } c \text{ ELSE } c'] = (I_{V-FV(c)} \wedge \bar{AG}[e]) \vee (I_{V-FV(c')} \wedge \bar{AG}[e'])$$

where $V=\text{FV}(\text{IF } b \text{ THEN } c \text{ ELSE } c')$.
\[ G[c;c'] \equiv (I_{V-FV(c)} \land G[c]) \cdot (I_{V-FV(c')} \land G[c']) \]

where \( V = FV(c;c') \).

The case of iteration is more complicated and uses the ability to encode finite sequences of finite parts of states. If the body of the loop is executed \( n \) times then it gives rise to \( n+1 \) states \( s_0, \ldots, s_n \), \( s_0 \) satisfying

\[ \neg b(s_0) \]

and

\[ b(s_{i+1}) \text{ and } <c,s_i,s_{i+1}> \text{ for } 0 \leq i < n. \]

The finite part of these states corresponding to the set \( FV(\text{WHILE } b \text{ DO } c) \) can now be encoded as a single vector of elements. So define \( \Theta(X,i,Y) \) to be the formula

\[ \bigwedge_{j=1}^{m} \Theta_{k_j}^{R_j}(x_j,i,y_j) \]

where \( FV(\text{WHILE } b \text{ DO } c) = \{x_1, \ldots, x_m\} \) (=V), \( x_j \) has sort \( k_j \) and sequences of elements of sort \( k_j \) are encoded as elements of sort \( R_j \). Furthermore, \( X \) is a vector of the variables of \( V \) and \( Y \) is a vector of distinct new variables \( y_1, \ldots, y_m \) of sorts \( R_1, \ldots, R_m \), respectively.

The formula \( G[\text{WHILE } b \text{ DO } c] \) is now defined as follows:

\[ \exists n. \exists Y. (\Theta(\overline{x},n,Y) \land \Theta(X,0,Y) \land b \land (\forall i. \forall x'. (i < n \land \Theta(x',i+1,Y)) \Rightarrow (\exists x''. b_x' \land (G[c] \land I_{V-FV(c)} x'' x' \land \Theta(x'',i,Y))))). \]

It is straightforward but tedious to prove that with this definition of \( G[c] \) the expressiveness condition is fulfilled. We omit the details. //
In this chapter we have presented a proof system for total correctness of while programs. As we required in Chapter 1 the formulas $P(c)Q/V$ of the proof system has post-conditions that express relationships between initial and final states. When developing the proof system we have, to a large extent, used ideas from existing proof systems, especially from those suggested by Manna and Pnueli in /MaPn74/ and Harel in /Ha79/. We shall in this section discuss some of the important differences between our proof system and those presented in the literature.

Manna and Pnueli seem to be the first to let post-conditions express relations between states /MaPn74/. The same idea has been used by Jones in /Jo80/. However, the rules of their proof systems seem rather heavy compared with for instance those of /Ha79/ where the post-conditions are predicates on (single) states. As an example consider the rule for the while loop. Here /MaPn74/ gives a general rule of the following form (using the notation of the previous sections)

$$P(ab)c>Q_A(\neg b\nu>\bar{u}), \neg X.X.Q A \rightarrow P,$$

$$\forall X.Y.X.Q_A X \rightarrow Q, \forall X.P A \rightarrow Q X$$

where $u$ is a term whose value is in a well-founded set and $>$ is the ordering on that set. On the other hand, the rule of /Ha79/ can be written as

$$P(0)\rightarrow b, P(z+1)A b(c)P(z)$$

$$\exists z.P(z)<WHILE b DO c>P(0)$$

where $z$ is a variable of sort nat. The rule given in /Jo80/ has a long list of assumptions (as that of /MaPn74/) and this seems to
motivate Jones's writing (/Jo80 p113/):

A reservation about the use of post-conditions of state pairs is the greater length of the list of properties which must be proved about them.

Aczel shows in /Ac82/ that by introducing appropriate abbreviations the apparent complexity of the rules can be reduced. In Aczel's work pre- and post-conditions of the correctness formulas are not formulas of some assertion language but sets of states and relations between states, respectively. His rule for iteration has the form

\[ Pab(c)PAQ \]
\[ P(\text{WHILE } b \text{ DO } c)Q^b \]

where \( Q \) is a transitive (that is, \( sQs' \) and \( s'Qs'' \) implies \( sQs'' \)) and well-founded (that is, if \( s_0, s_1, \ldots \) satisfies \( s_1Qs_{i+1} \) then the state sequence is finite) relation. Furthermore, \( Q^b \) is the reflexive closure of \( Q \) (that is, \( sQ^b s' \) if and only if either \( sQs' \) or \( s=s' \)).

To some extent we have adopted Aczel's abbreviations. Since we have pre- and post-conditions to be formulas of some assertion language we have syntactic versions of his semantic abbreviations:

for instance, \( _1Q_2 \) is defined relative to a finite set \( V \) of variables and means \( \exists x'. (Q_{1x'} \land Q_{2x'}^x) \) (for appropriate vectors \( X, X' \) and \( x' \)) and it corresponds closely to the usual relational composition used in /Ac82/.

So our conclusion is that compared with Harel's rule /H/ our rule /WHILE-/ may still seem clumsy but it is much more readable than that suggested by Manna and Pnueli, /MP/.

Let us now comment on the role of the component \( V \) of the formulas \( P(c)Q/V \). It has been introduced in order to express the post-condi-
tions of the formulas sufficiently precisely. To illustrate the problem further consider the assignment statement \( x := e \). Only the value of \( x \) is changed by executing this statement and the state transformation performed by the statement is given by the relation

\[
\{(s, s_x^{e(s)}) | s \text{ is a state}\}.
\]

However, this cannot be expressed by a formula in our (first-order) assertion language (we have assumed that we have an infinite number of program variables). We have solved the problem by introducing the component \( V \) in the formulas and letting the post-condition of the assignment axiom express the following relation

\[
\{(s, s') | x(s') = e(s) \text{ and if } y \in V - \{x\} \text{ then } y(s) = y(s')\}.
\]

The completeness result of Section 2.4 shows that this is indeed sufficient. Another possibility, of course, would be to restrict ones attention to a (global) finite set of program variables and then parameterise the proof system on that set. The states will then have finite domains and the expressibility problem above disappears. The problem does not receive any attention in /MaPn74/ and is not relevant for /Ac82/ and /Jo80/ where post-conditions are relations.

As for a general comparison with other proof systems for total correctness we have already mentioned that our axioms and rules are motivated by those suggested by Manna and Pnueli in /MaPn74/ and Harel in /Ha79/. In fact, the rules /IF-1/, /cons-T/ and /inv-T/ are derived from the rules of /MaPn74/ and we shall not comment further on them.

The rule for the while loop, /WHILE-T/, ensures termination of the loop essentially as suggested by /Ha79/ (the rule /H/ above): intuitively, the variable \( z \) counts the (maximal) number of times the
body of the loop has to be executed yet. A variant of this rule was originally suggested by Knuth in /Kn73/ and the idea was to count the number of times the body already has been executed and then bound this counter initially. The idea has later been used in for instance /So77a/ and /LuSu77/.

Except for the way termination of the loop is ensured the rule /WHILE-\*J/ has some resemblance with that of /MaPn74/ (/MP/) and that of /Ac82/ (/A/). In /WHILE-\*J/ we require that $P(z) \wedge R \wedge I \Rightarrow Q$ and $Q' \Rightarrow Q$ hold ($Q$ describes the effect of executing the loop, $Q'$ that of the body). So essentially, the relation specified by $Q$ is the transitive reflexive closure of that specified by $Q'$. In the rule /MP/ the effects of executing the body and the loop are described by the same formula, that is, $Q$ and $Q'$ are equal and can be thought of as representing the transitive reflexive closure of the effect of executing the body of the loop. The rule /A/ expresses the relationship more directly: $Q$ is the transitive reflexive closure of $Q'$.

Finally, let us discuss the soundness and completeness proofs of the sections 2.3 and 2.4, respectively. First, the soundness proof follows the same pattern as usually. On the other hand, the completeness proof seems to be non-standard: we have proved that the proof system is complete in the sense of Cook. This is the usual approach for proof systems for partial correctness but for proof systems for total correctness one usually follows Harel /Ha79/ and proves arithmetical completeness. As we have seen the completeness result "in the sense of Cook" is strong enough to give the arithmetical completeness result as well but apart from this it is unclear whether anything has been gained, that is, whether there are any interesting data types with models that are not arithmetical but where the
expressiveness condition is fulfilled. Some work related to this has already been done, especially in the context of proof systems for partial correctness (/BeTu80/, /Li77/).

As mentioned in Section 2.4 we are using an expressiveness condition equivalent to the usual one requiring the strongest post-conditions to be expressible. In the proof of the completeness result we have no need for additional variables except one (z) for the while loop. This is contrary to the completeness proof for the usual proof systems (for partial as well as total correctness) where in certain cases there are introduced new variables to take snapshots of the program variables. This trick is not needed here and the reason seems to be that we always will have access to both the initial and the final values of the variables because of the distinction between shadow and program variables.

In the completeness proof for the while loop we modify the body of the loop to count the number of unfoldings that are allowed. This "programming with the natural numbers" is not needed in the usual proof for arithmetical completeness (see for instance /Ha79/) where the termination condition P(z) is constructed using the ability to code finite sequences of elements as a single element - just as in our proof of Lemma 2.4-1. In fact, the proof of this lemma can also be found in /Ha79/.
3 Run-Time Analysis of While Programs

The aim of this chapter is to extend the proof system for total correctness of the previous chapter to prove run-time properties as well. We have already in Section 1.3 sketched how this will be done: the formulas $P(c)Q/V$ of the total correctness proof system are extended with a third formula $R$ of the assertion language expressing a property of the run-time of the program $c$, and furthermore, the axioms and rules are extended with deductions about these time formulas.

When analysing the run-time of a program it will always be on the basis of some computational model. In Section 3.1 we define a computational model for a data type and we show how it can be used to specify the semantics and the run-time requirements of the while programs. The actual proof system, called $\mathcal{R}$, for proving run-time properties of while programs is presented in Section 3.2. As for the total correctness proof system of Chapter 2 we can, under certain assumptions, obtain soundness and completeness results for $\mathcal{R}$. These results are presented in the sections 3.3 and 3.4, respectively. Pragmatically, we are interested in how naturally existing informal run-time analyses can be formulated in the proof system. To discuss this we shall in Section 3.5 consider the bubble sorting algorithm. In Section 1.2 we reviewed a couple of suggestions for how to formalise run-time analyses of programs within a logical formalism. We shall in Section 3.6 see how these ideas can be reformulated in our framework.
and we shall argue that although they are at least as powerful as the proof system $\mathcal{A}$ from a theoretical point of view we shall prefer $\mathcal{A}$ since it gives rise to more natural formalisations of the informal analyses. Finally, Section 3.7 contains, among others, a comparison with the approaches for analysing loop programs reviewed in Section 1.2.

3.1 RUN-TIME REQUIREMENTS OF THE WHILE LANGUAGE

As mentioned in Chapter 2 the idea is that the algorithms operate on data structures specified by data types. The implementation of the data structures and the operations on them determine the run-time requirements of the algorithms. One aspect of the implementation has already been captured by the definition of a model for the data type in Section 2.1 namely that of the meaning of the operations. We shall now extend the model to specify the run-time requirements of the operations of the data type as well.

A DATA TYPE AND ITS COMPUTATIONAL MODEL

Syntactically, a data type is specified by a set $K$ of sorts and a $K$-sorted signature $\Sigma$. The semantics of the data type is specified by a model for the $K$-sorted signature (see Section 2.1). The semantics and the run-time requirements of the data type will now be specified by a computational model for it.

A computational model for the data type is a model $\mathcal{M}$ for it extended with the following specifications of cost-functions:

- for each sort $k$ of $K$ there is a total function $^+ \mathcal{M}_k$ of functionality $\mathcal{M}_k \rightarrow \mathbb{N}$.
- for each function symbol $f$ of arity $(k_1 \ldots k_m, k)$ there is a total function $f^+$ of functionallity $\mathcal{M}_{k_1} \times \cdots \times \mathcal{M}_{k_m} \rightarrow \mathbb{N}$.
- for each relation symbol $p$ of arity $k_1 \ldots k_m$ there is a total function $p^+$ of functionallity $\mathcal{M}_{k_1} \times \cdots \times \mathcal{M}_{k_m} \rightarrow \mathbb{N}$.

The function $\gamma^+$ specifies the time required to fetch an element of the set $\mathcal{M}_k$. The cost-functions $f^+$ and $p^+$ associated with the function symbol $f$ and the relation symbol $p$, respectively, will specify the time required in order to perform the given operation on a given tuple of values.

A numerical computational model is a computational model with an underlying numerical model (see Section 2.3). An arithmetical computational model is a computational model with an underlying arithmetical model (see Section 2.4).

Example 3.1-1: As an example of a computational model we shall extend the standard model of Peano Arithmetic (Example 2.1-2) to specify a computational model. The cost-function $\gamma^+$ associated with the sort nat will be the constant function 1, that is, $n^+ = 1$ for every natural number $n$. The cost-functions associated with the symbols $+$, $\cdot$, $\equiv$ and $\prec$ are all the constant function 1, so for instance $(+)^+(n,n') = 1$ and $(\cdot)^+(n,n') = 1$ for every pair $(n,n')$ of natural numbers.

This computational model is based on the so-called uniform cost criterion where the idea is that every operation (including that of fetching the data) requires one unit of time (see for instance /AHU74/). Therefore we shall call it the uniform computational model for Peano Arithmetic.

Given a computational model for a data type and given a state s we can for each term e and boolean expression b determine the time
required to evaluate it in that state. The time $e^*(s)$ required to evaluate $e$ in $s$ is given by the table

<table>
<thead>
<tr>
<th>$e$</th>
<th>$e^*(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x(s)^+$</td>
</tr>
<tr>
<td>$v$</td>
<td>$v^+$</td>
</tr>
<tr>
<td>$f(e_1,\ldots,e_m)$</td>
<td>$e_1^<em>(s)+\ldots+e_m^</em>(s)+f^*(e_1(s),\ldots,e_m(s))$</td>
</tr>
</tbody>
</table>

The time $b^*(s)$ required to evaluate $b$ in $s$ is given by the table

<table>
<thead>
<tr>
<th>$b$</th>
<th>$b^*(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE (resp. FALSE)</td>
<td>0</td>
</tr>
<tr>
<td>$p(e_1,\ldots,e_m)$</td>
<td>$e_1^<em>(s)+\ldots+e_m^</em>(s)+p^*(e_1(s),\ldots,e_m(s))$</td>
</tr>
<tr>
<td>$b_1\land b_2$ (resp. $b_1\lor b_2$)</td>
<td>$b_1^<em>(s)+b_2^</em>(s)$</td>
</tr>
<tr>
<td>$b_1 \rightarrow b_2$, $b_1 \leftrightarrow b_2$</td>
<td>$b_1^<em>(s)+b_2^</em>(s)$</td>
</tr>
</tbody>
</table>

This shows that the logical connectives are free in the sense that they require no time to be evaluated. This reflects the idea that the important operations are those of the data type. If we are especially interested in the operations on the boolean values then we will have to introduce a sort, say bool, in the data type and include symbols for the various logical connectives in the signature.

To illustrate how this framework can be used we shall close this subsection with a couple of examples that will be used in the worked examples to be presented later.

Example 3.1-2: As the first example we shall extend the data type of Peano Arithmetic to specify constants corresponding to the natural numbers and to have some additional operations. So we have one sort, nat, and the {nat}-sorted signature specifies the following symbols
- for each natural number $n$ there is a symbol (ambiguously written $n$) of sort $\text{nat}$,
- $+, -, \cdot$ and $/$ are function symbols of arity $(\text{nat nat nat})$,
- $=, <, \leq, >$ and $\geq$ are relation symbols of arity $\text{nat nat}$.

We shall call this data type the data type of Extended Peano Arithmetic.

The standard model of Peano Arithmetic (Example 2.1-2) can easily be extended to specify a model for the new data type called the standard model of Extended Peano Arithmetic. The symbol $n$ is interpreted as the natural number $n$, the symbols $-$ and $/$ are interpreted as subtraction and division of natural numbers, respectively. (In order to avoid the extra complications that might arise by the introduction of an error element with the sort $\text{nat}$ we shall assume that $n-n'=0$ for $n'>n$ and that $n/0=0$.) The new relation symbols are given their obvious implementations as the relations "less than or equal to", "greater than" and "greater than or equal to", respectively.

The uniform computational model for the Extended Peano Arithmetic is the straightforward extension of that of Example 3.1-1 for Peano Arithmetic. The cost-function associated with the sort $\text{nat}$ is the constant function $1$ and so are the cost-functions associated with the function symbols and the relation symbols.

Example 3.1-3: The framework presented in Section 2.1 allows us to write while programs operating on a variety of data structures. We shall in this example specify a data type of one-dimensional arrays. The syntax of the while language suggests that we regard an array as one entity rather than a set of indexed entities but, as we shall see, this need not be reflected in a computational model for the data type.
The data type of one-dimensional arrays have two sorts, \textit{nat} and \textit{array}. The \{\textit{nat},\textit{array}\}-sorted signature specifies the following symbols in addition to those of the data type of Extended Peano Arithmetic (Example 3.1-2):

- \textit{init} is a function symbol of arity \( (\textit{nat},\textit{array}) \),
- \([\cdot] \) is a function symbol of arity \( (\textit{array} \ \textit{nat},\textit{nat}) \),
- \textit{upd} is a function symbol of arity \( (\textit{array} \ \textit{nat} \ \textit{nat},\textit{array}) \),
- \textit{length} is a function symbol of arity \( (\textit{array},\textit{nat}) \), and
- \( = \) is a relation symbol of arity \( \textit{array} \ \textit{array} \).

A simple example program is the composite statement

\[
x: = a[i]; \ a: = \text{upd}(a,i,a[j]); \ a: = \text{upd}(a,j,x)
\]

where \( a \) is a variable of sort \textit{array} and \( x, \ i \) and \( j \) are variables of sort \textit{nat} (the intention being that the values of the two entries \( i \) and \( j \) of the array \( a \) are swapped).

We shall now extend the uniform computational model for the Extended Peano Arithmetic (Example 3.1-2) to specify a computational model, say \( \mathcal{m} \), for the new data type. We shall let \( \mathcal{m}_\textit{array} \) be the set of finite sequences of natural numbers. For any such sequence \( w \) we define \( w^+ \) to be the constant 0. The interpretations of the new symbols are as follows:

- \( \text{init}(n) = 0\ldots0 \ (n \ 0's) \),
- \( n_1\ldots n_m[i] = \begin{cases} n_i & \text{if } 1 \leq i \leq m \\ 0 & \text{otherwise} \end{cases} \),
- \( \text{upd}(n_1\ldots n_m,i,j) = \begin{cases} n_1\ldots n_{i-1} j n_{i+1}\ldots n_m & \text{if } 1 \leq i \leq m \\ n_1\ldots n_m & \text{otherwise} \end{cases} \),
- \( \text{length}(n_1\ldots n_m) = m \),
- \( = \) is interpreted as the identity relation on \( \mathcal{m}_\textit{array} \).

The time requirements of these operations are as follows:
- $\text{init}^+(n) = n$ for $n \in \text{nat}$.
- $w[i]^+ = 1$ for $w \in \text{array}$, $i \in \text{nat}$.
- $\text{upd}^+(w, i, j) = 2$ for $w \in \text{array}$, $i, j \in \text{nat}$.
- $\text{length}^+(w) = 2$ for $w \in \text{array}$.
- $(=)^+(w, w') = \text{length}(w) + \text{length}(w')$ for $w, w' \in \text{array}$.

The computational model reflects that we are thinking of an implementation of the data type where we have access to the individual elements of an array rather than the complete array. The statement $a := \text{upd}(a, i, j)$ will for instance correspond to the Pascal statement $a[i] := j$ both semantically and with respect to the run-time requirements of some computational model based on the uniform cost criteria: the operation $\text{upd}(a, i, j)$ requires two units of time because one unit is needed to find the $i$'th entry of the array $a$ and one unit is needed for the assignment. The assignment to $a$ in $a := \text{upd}(a, i, j)$ is free and in some sense this reflects that we have no access to the complete array.

### Semantics and Run-Time Requirements

Given a model for the data type, the semantics of the while programs is given by an "initial state - finite state" relation $\langle c, s \rangle \rightarrow \langle s' \rangle$. We shall now extend it to specify the run-time as well, more precisely, given a computational model for the data type we shall define a relation $\langle c, s \rangle \xi \rightarrow \langle s' \rangle$ with the intuitive meaning that if the execution of the program $c$ starts in the state $s$ then it will terminate after exactly $r$ time units and the final state will be $s'$. The set $\mathcal{S}$ of axioms and rules given in the table below is the appropriate extension of those of $\mathcal{S}_0$. 

83
Semantics and run-time of while programs: \( \$ \)

The assignment axiom \( /\text{ass-} \$ / \) reflects the assumption that the time taken to store a value is the same as that taken to fetch it. This might not be appropriate and then the computational model can easily be modified to specify costs for storing values of the various sorts. In the rules for the conditional \( /\text{IF-} \$ / \) and for the iterative statement \( /\text{WHILE-} \$ / \) we have ignored the possible time requirements for a jump following a test. Again the computational model can easily be modified to specify the costs of such jumps and the rules of \( \$ \) changed accordingly.
The formal system \( \mathcal{G} \) corresponds very closely to that, \( \mathcal{G}_0 \), given in Section 2.1 for the specification of the semantics of the while programs. In fact, we have for every program \( c \) and pair \((s,s')\) of states that
\[
\langle c,s \rangle \rightarrow s' \quad \text{(in } \mathcal{G}_0) \]
if and only if
\[
\langle c,s \rangle \xrightarrow{r} s' \quad \text{(in } \mathcal{G}) \]
for some natural number \( r \).

So it is not surprising that we have the following three lemmas corresponding to those of Section 2.1 (and they are proved in the same way):

**Lemma 3.1-1:** If \( \langle c,s \rangle \xrightarrow{r} s' \) and \( \text{VNFV}(c) = \emptyset \) for a finite set \( V \) of variables then \( s \models_V s' \).

**Lemma 3.1-2:** If \( \langle c,s \rangle \xrightarrow{r} s' \) and \( s \models_V s_0 \) for a finite set \( V \) of variables satisfying \( \text{FV}(c) \subseteq V \) then \( \langle c,s_0 \rangle \xrightarrow{r} s_0' \) for some state \( s_0' \) with \( s_0' \models_V s' \).

**Lemma 3.1-3:** If \( \langle c,s \rangle \xrightarrow{r} s' \) and \( \langle c,s \rangle \xrightarrow{r'} s'' \) then \( s'' = s'' \) and \( r = r' \).

### 3.2 THE PROOF SYSTEM \( \mathcal{R} \)

Having defined the computational models for the data types we shall now extend the proof system \( \mathcal{F} \) of Chapter 2 to prove run-time properties of while programs. In order to express the required properties of the run-time we shall introduce a new sort of formulas of the assertion language called time formulas. A time formula is a formula \( R \) of the assertion language that as free variable has the very special variable \( \text{time} \) and otherwise only program variables are allowed as free variables in \( R \). (So we have the extended set \( X' \) of variables to be \( X \cup \text{time} \),
The intuitive idea is that the variable `time` which has sort `nat` denotes the run-time of some program whereas the other variables refer to the initial state of the computation. Thus `time` will neither be a program nor a shadow variable. We shall write \( \mathcal{R}(s,r) \) for the truth of \( R \) in the state \( s \) and with \( r \) being the value of the variable `time`.

The formulas of the proof system \( \mathcal{Q} \) for run-time analysis of while programs are going to have the form \( P(c:R)Q/V \) where \( P, c, Q \) and \( V \) are as in the proof system \( \mathcal{T} \) for total correctness and \( R \) is a time formula. However, before presenting the axioms and rules of the proof system we shall need some preliminaries.

**The Time Expressiveness Condition**

When constructing the proof system for run-time analysis we shall need time formulas corresponding to the cost-functions of the computational model. We shall therefore impose a condition that ensures that these formulas exist.

Given a data type specified by a set \( K \) of sorts and a \( K \)-sorted signature \( \Sigma \), and given a computational model \( \mathcal{M} \) for it, the time expressiveness condition is fulfilled if

- for each sort \( k \) of \( K \) there is a time formula \( E^+(x) \) with the free variable \( x \) of sort \( k \) and such that for every pair \( (s,r) \)
  \[ E^+(x)(s,r) \text{ if and only if } x(s)^+ = r, \]

- for each function symbol \( f \) of arity \( (k_1, \ldots, k_m, k) \) there is a time formula \( E_f(x_1, \ldots, x_m) \) with the free variables \( x_1, \ldots, x_m \) of sorts \( k_1, \ldots, k_m \), respectively, and such that for every pair \( (s,r) \)
∀E_p(x_1, ..., x_m)(s, r) if and only if f^+(x_1(s), ..., x_m(s)) = r,

- for each relation symbol p of arity k_1, ..., k_m there is a time formula E_p(x_1, ..., x_m) with the free variables x_1, ..., x_m of sorts k_1, ..., k_m, respectively, and such that for every pair (s, r):

∀E_p(x_1, ..., x_m)(s, r) if and only if p^+(x_1(s), ..., x_m(s)) = r.

It is important to note that the time expressiveness condition is a property of the data type and its computational model, and as such it is independent of the programming language defined on top of the data type.

Example 3.2-1. It is easy to check that the following data types and computational models satisfy the time expressiveness condition:

- the data type of Peano Arithmetic and its uniform computational model (Example 2.1-1 and Example 3.1-1),
- the data type of Extended Peano Arithmetic and its uniform computational model (Example 3.1-2),
- the data type of one-dimensional arrays and its computational model (Example 3.1-3).

The time expressiveness condition ensures that we have time formulas expressing the exact time requirements of the terms and boolean expressions. Before showing how these formulas can be obtained we shall introduce the following abbreviation: Given two time formulas R and R' we shall write R&R' for the formula

∃time'.∃time''.time=corea(time)+(time)time'.time(time' time)

Note that R(s, r) and R'(s, r') implies R&R'(s, r+r') and if R&R'(s, r) then r=r'+r'' for some r' and r'' and furthermore R(s, r') and R'(s, r'').
Using this notation we can define a time formula \( E(e) \) for each term \( e \). Intuitively, \( E(e) \) is a formula for the exact time required to evaluate \( e \).

\[
\begin{array}{|c|c|}
\hline
e & E(e) \\
\hline
x & E^+(x) \\
v & E^+(v) \\
f(e_1, \ldots, e_m) & E(e_1) \oplus \ldots \oplus E(e_m) \oplus E(f(e_1, \ldots, e_m)) \\
\hline
\end{array}
\]

Furthermore, define \( E^S(e) \) to be the formula \( E(e) \oplus E^+(e) \). Intuitively, \( E^S(e) \) is a formula for the exact time required to evaluate \( e \) and then store its value. For each boolean expression \( b \) we get a time formula \( E(b) \) for the time required to evaluate it:

\[
\begin{array}{|c|c|}
\hline
b & E(b) \\
\hline
\top (\text{resp.} \ \bot) & \text{time=0} \\
\neg b & E(e_1) \oplus \ldots \oplus E(e_m) \oplus E(f(e_1, \ldots, e_m)) \\
b \lor b' (\text{resp.} \ b \land b', \ b \Rightarrow b', \ b \Leftarrow b') & E(b) \oplus E(b') \\
\hline
\end{array}
\]

Note that \( \text{FV}(E(e)) \subseteq \text{FV}(e) \), \( \text{FV}(E^S(e)) \subseteq \text{FV}(e) \) and \( \text{FV}(E(b)) \subseteq \text{FV}(b) \). It is straightforward to prove the following result:

**Lemma 3.2-1.** If the time expressiveness condition holds for the data type and its computational model then for every term \( e \), boolean expression \( b \) and pair \((s,r)\) of state and natural number

- \( \Phi E(e)(s,r) \) if and only if \( e^S(s)=r \),
- \( \Phi E^S(e)(s,r) \) if and only if \( e^S(s)+e(s)^+=r \),
- \( \Phi E(b)(s,r) \) if and only if \( b^S(s)=r \).
THE PROOF SYSTEM

As mentioned earlier the formulas of the proof system $\mathcal{R}$ have the form $P(c:R)Q/V$ where $P$, $c$, $Q$ and $V$ are as in the proof system $\mathcal{T}$ of Chapter 2 and $R$ is a time formula. As in Section 2.2 we shall impose a well-formedness condition on the formulas in order to ensure that $V$ contains the program variables we are interested in. So we say that the formula $P(c:R)Q/V$ is a well-formed formula of $\mathcal{R}$ if

- $FV(c) \subseteq V$,
- $FV(P) \subseteq V \cup \overline{V}$ and $FV(R) \subseteq \text{vol}(\text{time})$.

The validity of a well-formed formula (written $P(c:R)Q/V$) is now defined as follows

for every state $s$ satisfying $F \equiv (s)\,$ there is a state $s'$ and a natural number $r$ such that

$$<c, s> \xrightarrow{s'} Q(s, s') \land R(s, r).$$

That is, if the pre-condition $P$ holds then the program will terminate, the initial state and the final state satisfy the post-condition $Q$ and the run-time satisfies the time formula $R$.

To simplify the notation used in the axioms and rules below we shall introduce yet another abbreviation. For a relational formula $Q$ and a time formula $R$ we shall write $Q'R$ for the time formula $\exists X'. Q'_{X'}$, where $X'$ is a vector of the program variables of some finite set $V$ (with $FV(Q) \subseteq V \cup \overline{V}$ and $FV(R) \subseteq \text{vol}(\text{time})$), $\overline{X}$ is the corresponding vector of shadow variables and $X'$ is a vector of distinct new variables of appropriate sorts and of the same length as $X$. For any pair $(s, r)$ of state and natural number we have

$$FQ'R(s, r) \text{ if and only if } FQ(s, s') \land FR(s', r) \text{ for some state } s'.$$
Assuming that the time expressiveness condition is fulfilled for the data type and its computational model we have the following set of axioms and rules in the proof system $\mathcal{R}$.

The proof system $\mathcal{R}$

\[
\begin{align*}
\text{/ass-} & \mathcal{R}/ & P(x = e : E) & S(e) \rightarrow x = e \rightarrow V \\
\text{/IF-} & \mathcal{R}/ & P \{ c \rightarrow R \} \rightarrow Q \rightarrow V, P \{ c \rightarrow R \} \rightarrow Q \rightarrow V \\
\text{/;} & \mathcal{R}/ & P \{ c_1 : R_1 \} \rightarrow P \{ c_2 : R_2 \} \rightarrow Q \rightarrow V \\
\text{/WHILE-} & \mathcal{R}/ & P \{ z+1 \} \rightarrow P \{ z \} \rightarrow Q \rightarrow V, P \{ z \} \rightarrow R, Q \rightarrow Q, R \rightarrow R \\
\text{/cons-} & \mathcal{R}/ & P \rightarrow P', P \{ c : R \} \rightarrow Q \rightarrow V, Q' \rightarrow Q, R' \rightarrow R \\
\text{/inv-} & \mathcal{R}/ & P \{ c : R \} \rightarrow Q \rightarrow V \\
\end{align*}
\]

We shall write $\mathcal{R} \vdash P \{ c : R \} \rightarrow Q \rightarrow V$ if the formula $P \{ c : R \} \rightarrow Q \rightarrow V$ is provable using the axioms and rules above with the constraint that all formulas of the assertion language must be true and all the formulas of $\mathcal{R}$ must be well-formed.

First note that the axioms and rules of $\mathcal{R}$ are extensions of those of $\mathcal{T}$ (see Section 2.2); in fact, we have just added information about
the run-time requirements. In /ass-/ we have recorded that $E^S(e)$ describes the time required to evaluate and store the value of $e$. In /IF-/ we have given a time formula $R$ holding for both branches of the construct and we simply "add" the time for the test, that is, $E(b)$. That we have the same time formula for both branches is no restriction, it can always be obtained by applying the rules /cons-\exists/ and /inv-\exists/.

The rule /;\exists/ is quite complicated. The formula $R_2$ holds for the run-time of $c_2$ relative to the initial state of $c_2$. Since $Q_1$ describes the effect of executing $c_1$ and since the final state of $c_1$ is the initial state of $c_2$ we get that $Q_1 \cdot R_2$ is a property of the run-time of $c_2$ relative to the initial state of $c_1$. The formula $R_1$ is a property of the run-time of $c_1$ relative to its initial state so $R_1 \otimes (Q_1 \cdot R_2)$ is a property of the run-time of $c_1 ; c_2$. This is illustrated on the following figure:

In /WHILE-/ the time formula $R$ is a sort of invariant. If the body of the loop is not executed at all then the assumption $P(z) \land b \land E(b) \Rightarrow R$ ensures that $R$ holds. If the body of the loop is executed then the time for the test satisfies $E(b)$, the time for the body $R'$ and the run-time for the rest of the loop satisfies $R$ (by the assumption that it is a sort of invariant). As for the rule /;\exists/ we
get that the time requirements for the loop will satisfy 
\(E(b)R'(Q'R)\) since the effect of executing the body of the loop
once is described by \(Q'\). Since \(E(b)R'(Q'R)\to R\) holds by assumption
we get that \(R\) holds for the loop. This is illustrated on the following
figure:

The rules /cons-\(R\)/ and /inv-\(R\)/ should be straightforward to under-
stand.

**Example**

As an example of the use of the proof system \(R\) let us sketch a proof
of the formula

\[y=1\text{WHILE } y<x \text{ DO } (y:=y+1; \text{ fac:=y'fac}); \text{time(11'x+4)TRUE/\{x,y,fac\}.}\]

We shall use the data type of Peano Arithmetic (Example 2.1-1) and
its uniform computational model (Example 3.1-1). In the following let
\(V\) be an abbreviation for the set \(\{x,y,fac\}\).

We shall present the proof in a bottom up manner. To a large extent
the pattern of reasoning follows that of the example proof in the proof
system \(J\) given in Section 2.2 where we construct a proof of the for-
mula
So we shall mainly be concerned with the deductions about time formulas.

First we shall construct a proof for the body of the loop. The invariant $P(z)$ for the loop is as in Section 2.2:

$$(x=0 \land z=0 \land y=1) \land (x>0 \land x=y+z).$$

Since we consider the uniform computational model for the data type we can without loss of generality assume that $E^S(y+1)$ is the formula $\text{time}=4$ and using /ass-\(R\)/ we therefore get a proof of

$$P(z+1) \land y<x \land \{y:=y+1, y=fac, z=x\} \land \{y=x+1, y=fac\}.$$  

It can easily be verified that

$$(x=y+1 \land z=y+1) \land (x,fac, z=x) \land (y=fac)$$

and trivially we have

$$P(z+1) \land x<y \land \text{time}=4$$

so using first /inv-\(R\)/ and then /cons-\(R\)/ we get a proof of

$$P(z+1) \land y<x \land \{y:=y+1, y=fac, z=x\} \land \{y=x+1, y=fac\}.$$  

Similarly, we can construct a proof of the formula

$$P(z) \land \{fac:=y, fac:=y\} \land \{x, y, z\} \land \{x, y, z\}.$$  

These two proof can now be put together using the rule \(;\)-\(R\). We have

$$\text{INV}(I_{x, y, z} \land y=x+1, \land (P(z) \land y=x+1)) \Rightarrow P(z) \land x=x \land y=x+1$$

and

$$\text{time}=4 \Rightarrow (I_{x, y, z} \land y=x+1, \land \text{time}=4) \Rightarrow \text{time}=8$$

so using first \(;\)-\(R\)/ and then /cons-\(R\)/ we get a proof of

$$P(z+1) \land y<x \land \{y:=y+1, fac:=y\} \land \{x, y, z\} \land \{x, y, z\}.$$
In order to get a proof for the while loop we shall apply the rule \(\text{WHILE-/cs/}\). We have

\[ P(0) \to \neg(y < x). \]

The effect of executing the loop is chosen to be given by the formula \(\text{TRUE}\). The time formula, \(R\), being the time invariant for the loop is chosen to be the formula

\[ 11'y + \text{time} < 11'x + 15. \]

It is then straightforward to verify that

\[ P(z) \land (y < x) \land I \to \text{TRUE}, \]

\[ (x = x \land y = y + 1) \to \text{TRUE} \to \text{TRUE}, \]

and

\[ P(z) \land (y < x) \land \text{time} = 3 \to R \]

and

\[ (\text{time} = 3) \land (\text{time} = 8) \land (x = x \land y = y + 1) \to R. \]

In the last two formulas we use that we, without loss of generality, can choose \(E(y < x)\) to be the formula \(\text{time} = 3\). So using \(\text{WHILE-cs/}\) we get

\[ \exists z. P(z) \langle \text{WHILE } y < x \text{ DO } (y := y + 1; \text{ fac} := y \text{ fac}) : R \rangle \to \text{TRUE} / V. \]

To get the required proof we shall first apply the rule \(\text{cons-R/}\) with

\[ y = 1 \to \exists z. P(z) \]

and then the rule \(\text{inv-R/}\). Since

\[ y = 1 \to \text{time} < 11'x + 4 \]

we get a proof of the required formula using \(\text{cons-R/}\).

3.3 THE SOUNDNESS THEOREM FOR \(\mathcal{R}\)

We shall in this section and the next one discuss the theoretical properties of soundness and completeness for the proof system \(\mathcal{R}\). The
proof system is constructed under the assumption that the time expressiveness condition is fulfilled and this is, of course, reflected in the results we obtain. As for soundness, we have the following result:

**The Soundness Theorem for \( \mathcal{R} \)**

Given a data type and a numerical computational model for it, if the time expressiveness condition is fulfilled then for every well-formed formula \( P(c:R)Q/V \) of \( \mathcal{R} \)

\[ \mathcal{R}\cdot P(c:R)Q/V \Rightarrow T\cdot P(c:R)Q/V. \]

So except for the addition of the time expressiveness condition the result is similar to that holding for the proof system \( T \) (see Section 2.3). The proof of the soundness result for \( \mathcal{R} \) is an extension of that for soundness result for \( T \). We show that the axiom is valid and that the rules preserve validity.

**Case /ass-\( \mathcal{R} \):** We shall prove that the axiom is valid, that is

\[ \forall P(c:R)Q/V \]

So assume that \( P(s) \) holds for some state \( s \). From /ass-\( \mathcal{J} \) we have

\[ \langle x:=e,s \rangle \rightarrow e(s)+e(s)^+ \rightarrow s \]

and as in the validity proof for /ass-\( \mathcal{J} \) we have \( \forall I_{V-\{x\}} \wedge x=e(s),s^x \). From Lemma 3.2-1 we get \( E^S(e)(s,e^S(s)+e(s)^+) \) and the required result follows.

**Case /IF-\( \mathcal{R} \):** We shall prove that the rule preserves validity so assume that

(1) \( \forall \exists Pab(c:R)Q/V \)

and
To prove
\[ \vdash P(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 : E(b)@R)Q/V \]
assume that \( P(s) \) holds for some state \( s \). In the case where \( b(s) \) holds we get from (1) that for some \( s' \) and \( r \)
\[ \langle c_1, s \rangle \rightarrow s', PQ(s, s') \text{ and } PR(s, r). \]
Using /IF-\( \triangleright \)/ we then get
\[ \langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s \rangle \rightarrow^{b(s)+r} s' \]
and from Lemma 3.2-1 we have \( E(b)(s, b^S(s)) \) and thereby
\( F(E(b)@R)(s, b^S(s)+r) \). This proves the required result. The case where \( \neg b(s) \) holds is similar. ///

Case /;\( \triangleright \)/: We shall now assume that

(1) \( \vdash P(c_1 : R_1, P'AQ_1)Q_1/V \)

and

(2) \( \vdash P'(c_2 : R_2, Q_2/V \)

and we have to prove that
\[ \vdash P(c_1 ; c_2 : R_1@Q_1 \cdot R_2)Q_1 \cdot Q_2/V. \]

So assume that \( P(s) \) holds for some state \( s \). From (1) we get that
\[ \langle c_1, s \rangle \rightarrow s', P'\cdot AQ_1(s, s') \text{ and } PR_1(s, r) \]
for some \( s' \) and \( r \). Then \( P'(s') \) holds and (2) gives that
\[ \langle c_2, s' \rangle \rightarrow^{r'} s'', PQ_2(s', s'') \text{ and } PR_2(s', r') \]
for some \( s'' \) and \( r' \). From /;\( \triangleright \)/ we then get
\[ \langle c_1 ; c_2, s \rangle \rightarrow^{r+r'} s''. \]
As in the soundness proof for /;\( \triangleright \)/ we get \( FQ_1 'Q_2(s, s'') \). From \( PQ_1(s, s') \)
and \( R_2(s',r') \) we get \( Q_1 R_2(s, r') \) and since \( R_1(s, r) \) we get
\( R_1 \cup (Q_1 R_2)(s, r+r') \). This proves the result.

Case /WHILE-\( \mathcal{R} \)/: Now we assume that

1. \( FP(z+1) \wedge b \wedge c: R' \rightarrow P(z) \wedge Q'/V \cup z \),
   \( FP(0) \rightarrow b \),
   \( FP(z) \wedge b \wedge I_V \rightarrow Q \),
   \( FQ' \rightarrow Q \),
2. \( FP(z) \wedge b \wedge E(b) \rightarrow R \)
and
3. \( F(E(b) \wedge R' \wedge (Q' \wedge R)) \rightarrow R \)
where \( z \) is a variable of sort \textit{nat} satisfying \( z \notin V \). We have to prove
\( \exists z. P(z) \langle \text{WHILE} b \text{ DO } c; R \rangle \rightarrow Q/V \).

Analogue to the soundness proof for /WHILE-\( \mathcal{T} \)/ it is sufficient to prove that if \( FP(z)(s^n) \) holds for some state \( s \) and natural number \( n \) then for some \( s' \) and \( r \)

\( \langle \text{WHILE } b \text{ DO } c; s \rangle \mapsto s' \), \( FQ(s, s') \) and \( FR(s, r) \).

The proof is by induction on \( n \).

If \( n=0 \) then \( FP(0)(s) \) holds and thereby \( Fb(s) \). From /WHILE-\( \mathcal{P} \)/ we get

\( \langle \text{WHILE } b \text{ DO } c; s \rangle \mapsto b^S(s) \rightarrow s \).

As in the soundness proof for /WHILE-\( \mathcal{T} \)/ we get \( FQ(s, s) \). From Lemma 3.2-1 we have \( FE(b)(s, b^S(s)) \) and since \( FV(E(b)) \notin V \) we have
\( FP(z) \wedge b \wedge E(b)(s^0, b^S(s)) \). So from (2) we get \( FR(s^0, b^S(s)) \) and since \( FV(R) \notin V \cup \text{time} \) (the conclusion of the rule is well-formed) we get
\( FR(s, b^S(s)) \). This proves (§) in the case where \( n=0 \).

97
Assume now that (§) holds for n=n' and we shall prove it for n=n'+1. So assume that \( P(z)(s_{n'1}) \) holds, that is \( P(z+1)(s_{n''}) \). If \( \forall \ P(s) \) holds as well then we proceed as in the case n=0 above. In the case where \( \forall \ P(s) \) holds we get from (1) that for some \( s'' \) and \( r' \)

\[
\langle c, s_{n'}, r' \rangle \rightarrow s'', \quad P(z) \land Q'(s_{n'}, s'') \quad \text{and} \quad R'(s_{n'}, r').
\]

As in the soundness proof for /WHILE-/ we get that \( P(z)(s_{n''}) \) (replacing application of Lemma 2.1-1 by Lemma 3.1-1) so the induction hypothesis gives

\[
\langle \text{WHILE } b \text{ DO } c, s'' \rangle \rightarrow s', \quad P(s'', s') \quad \text{and} \quad R(s'', r)
\]

for some \( s' \) and \( r \). Then /WHILE-/ gives

\[
\langle \text{WHILE } b \text{ DO } c, s_{n'} \rangle \rightarrow^{b(s_{n'})+r'+r} s'.
\]

Using Lemma 3.1-2 with \( z \notin \text{FV} (\text{WHILE } b \text{ DO } c) \) and \( s_{n'}, s \) we get for some \( s_0 \) with \( s_0 \in \text{V} \), we get

\[
\langle \text{WHILE } b \text{ DO } c, s_{n'} \rangle \rightarrow^{b(s_{n'})+r'+r} s_0.
\]

As in the soundness proof for /WHILE-/ we get \( P(s, s_0) \). Lemma 3.2-1 gives \( P(b)(s_{n'}, b(s_{n'})) \). We have \( R'(s_{n'}, r') \) and furthermore \( P'(s_{n'}, s'') \) and \( R(s'', r) \). So altogether we have

\[
P(b) \land R'(Q''R)(s_{n'}, b(s_{n'})+r'+r).
\]

But then \( R(s_{n'}, b(s_{n'})+r'+r) \) follows from (3) and since \( \text{FV}(R) \in \text{time} \) holds we get \( R(s, b(s_{n'})+r'+r) \) as required. This completes the proof of (§) and thereby the soundness of the rule /WHILE-/.

Case /cons-/: Straightforward and therefore omitted.

Case /inv-/: Straightforward and therefore omitted.

This completes the proof of the soundness result for /\text{R}/.
The completeness result for the proof system $\mathcal{I}$ for total correctness is obtained under an expressiveness condition ensuring essentially that the effect of executing a program can be expressed by a relational formula of the assertion language (see Section 2.4). We shall now extend this condition and require that we for each program have a time formula expressing its run-time requirements. Under this assumption we obtain a completeness result for $\mathcal{K}$ very similar to that holding for $\mathcal{I}$. Furthermore, we shall show that for the data types with arithmetical computational models the time expressiveness condition (of Section 3.2) will be sufficient to ensure that $\mathcal{K}$ is complete.

**The Expressiveness Condition**

Given a data type and a numerical computational model for it we shall say that the expressiveness condition for $\mathcal{K}$ is fulfilled if

- the time expressiveness condition is fulfilled,
- for every while program $c$ there is a relational formula $G[c]$ with $FV(G[c]) \subseteq FV(c) \cup FV(c)$ and satisfying that for every pair $(s,s')$ of states

  $$ G[c] \models (s,s') $$

  if and only if

  $$ \langle c, s \rangle \xrightarrow{r} s'' \text{ for some state } s'' \text{ with } s'' \models_{FV(c)} s' \text{ and some natural number } r, $$

- for every while program $c$ there is a time formula $E[c]$ with $FV(E[c]) \subseteq FV(c) \cup \{\text{time}\}$ and satisfying that for every pair $(s,r)$
of state and natural number

\[ \forall \mathbf{c} \in \mathfrak{I}(s,r) \]

if and only if

\[ \langle c,s \rangle \vdash s' \] for some state \( s' \).

Note that if the expressiveness condition for \( \mathfrak{R} \) is fulfilled then so is the time expressiveness condition and the expressiveness condition for \( \mathfrak{T} \) (defined in Section 2.4).

As we shall see later the expressiveness condition for \( \mathfrak{R} \) will be fulfilled by for instance the data type of Extended Peano Arithmetic and its computational model (see Example 3.4-1 later).

**THE COMPLETENESS RESULT AND ITS PROOF**

Using this notion of expressiveness we have the following result

The Completeness Theorem for \( \mathfrak{R} \)

Given a data type and a numerical computational model for it,

if the expressiveness condition for \( \mathfrak{R} \) is fulfilled then for every well-formed formula \( P(c:R \triangleright Q/V) \) of \( \mathfrak{R} \)

\[ \forall P(c:R \triangleright Q/V) \text{ implies } \mathfrak{R}-P(c:R \triangleright Q/V). \]///

The proof of this result is by structural induction on the program \( c \). The proof is an extension of that for The Completeness Theorem for \( \mathfrak{T} \) given in Section 2.4.

**Case \( x:=e \):** We assume that

\( 1) \ \forall P(x:=e:R \triangleright Q/V) \)

holds and we shall construct a proof of the formula in \( \mathfrak{R} \). From /ass-R/
we get a proof of
\[ P(x:=e; E(e))_{V-|x|}^{A} x=e/V \]
so using /inv-\( \mathcal{R} / \) we get a proof of
\[ P(x:=e; P\alpha E(e) \supset \neg P\alpha I_{V-|x|}^{A} x=e/V. \]
As in the completeness proof for \( T \), case \( x:=e \), we have
\[ P\alpha I_{V-|x|}^{A} x=e \rightarrow Q \]
and below we shall prove that
(2) \[ P\alpha E(e) \rightarrow R. \]
So using the rule /cons-\( \mathcal{R} / \) we get a proof of the required formula.

To prove (2) assume that \( P\alpha E(e)(s,r) \) holds for some pair \( (s,r) \) of state and natural number. From \( P(s) \) and (1) we get that
\[ (x:=e,s\rightarrow s', \ Q(s,s') \ and \ R(s,r')) \]
hold for some \( s' \) and \( r' \). From /ass-\( \mathcal{R} / \) and Lemma 3.1-3 we get that
\[ r'=e^S(s)+e(s)^+. \]
From \( P\alpha E(e)(s,r) \) and Lemma 3.2-1 we have \( r=e^S(s)+e(s)^+ \)
so \( R(s,r) \) holds. This proves (2).

Case IF \( b \) THEN \( c_1 \) ELSE \( c_2 \): Assume that
(1) \[ P(IF \ b \ THEN \ c_1 \ ELSE \ c_2 ;R \supset Q/V \]
holds and we shall construct a proof of the formula in \( \mathcal{R} \). Define \( R' \) to be the time formula
\[ (b\alpha E[c_1 \Box]) \supset (\neg b\alpha E[c_2 \Box]). \]
Using the expressiveness condition it is then straightforward to prove the validity of the two formulas
\[ P\alpha b< c_1 ;R' \supset Q/V \]
and
The induction hypothesis can then be applied and we get proofs of the two formulas in $\mathcal{R}$. Using the rule /IF/$\mathcal{R}$ we then get a proof of

$$P \langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 : E(b) \otimes R' \rangle \Rightarrow Q/V.$$ 

Below we shall prove that

(2) $\forall P \alpha (E(b) \otimes R') \Rightarrow R$

so by applying first /inv$/\mathcal{R}$ and then /cons$/\mathcal{R}$ we get the proof of the required formula.

To prove (2) assume that $P \alpha (E(b) \otimes R') (s, r)$ holds for some pair $(s, r)$ of state and natural number. From $FP(s)$ and (1) we get that

$$\langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 , s', s' \rangle \Rightarrow s', FQ(s, s') \text{ and } FR(s, r')$$

for some $s'$ and $r'$. We have two cases. If $P b(s)$ holds then /IF$/\mathcal{P}$ gives that $r' = b^S(s) + r''$ and

$$\langle c_1 , s \rangle \Rightarrow s''.$$ 

From $E(b) \otimes R' (s, r)$ we get that $r = r_0 + r_1$, $E(b)(s, r_0)$ and $R'(s, r_1)$ for some $r_0$ and $r_1$. Using the expressiveness condition for $\mathcal{R}$ and the definition of $R'$ this means that for some $s''$

$$\langle c_1 , s \rangle \Rightarrow s''.$$ 

Lemma 3.1-3 then gives that $r'' = r_1$ (and $s' = s''$). From $E(b)(s, r_0)$ and Lemma 3.2-1 we get that $r_0 = b^S(s)$ and thereby $r = r'$. So $FR(s, r)$ holds.

The case where $P b(s)$ holds is similar. 

Case $c_1 \uplus c_2$: We shall now assume that

(1) $P \langle c_1 \uplus c_2 : R \Rightarrow Q/V \rangle$

and we shall construct a proof of the formula in $\mathcal{R}$. As in the completeness proof for $\mathcal{J}$, case $c_1 \uplus c_2$, we define the formulas $Q_1$, $Q_2$ and
to be
\[ \begin{align*}
Q_1 &= \text{G}[c_1]_{V-FV(c_1)} \\
Q_2 &= \text{G}[c_2]_{V-FV(c_2)}
\end{align*} \]
and
\[ P' = \exists x'. \text{G}[c_2]_{x^2/x} \]
(where \(x, \bar{x}\), and \(x'\) are "as usual"). Furthermore, define the two time formulas \(R_1\) and \(R_2\) to be \(\text{E}[c_1]_{1}\) and \(\text{E}[c_2]_{0}\), respectively. A straightforward modification of the proofs of the similar results in the completeness proof for \(T\), case \(c_1; c_2\), shows that the formulas
\[ P < c_1: R_1 > P' A Q_1 / V \]
and
\[ P' < c_2: R_2 > Q_2 / V \]
are valid. The induction hypothesis then gives that they are provable in \(\mathcal{R}\). So using the rule \(/; -\mathcal{R}/\) we get a proof of
\[ P < c_1; c_2: R_1 \otimes (Q_1 \& R_2) > (Q_1 \& Q_2) / V. \]
As in the completeness proof for \(T\), case \(c_1; c_2\), we have
\[ \vdash \text{PAR}_{Q_1 \& Q_2} \to Q. \]
Below we shall prove that
\[ (2) \quad \vdash \text{PAR}_{Q_1 \& R_2} \to R \]
so using first \(/\text{inv-}\mathcal{R}/\) and then \(/\text{cons-}\mathcal{R}/\) we get a proof of the required formula in \(\mathcal{R}\).

To prove (2) assume that \(\vdash \text{PAR}_{Q_1 \& R_2}(s, r)\) holds for some pair \((s, r)\) of state and natural number. Then there are natural numbers \(r_1\) and \(r_2\) and a state \(s'\) such that \(\vdash R_1(s, r_1)\), \(\vdash Q_1(s, s')\), \(\vdash R_2(s', r_2)\) and \(r = r_1 + r_2\). Using the expressiveness condition this means that
\[ <c_1, s > \xrightarrow{r_1} s'_0 \quad \text{for some} \quad s'_0, \]
for some $s'$ with $s_1^{'} = s'$ and some $r'$ and

$$\langle c_2, s' \rangle \rightarrow_{2}^{r} s'$$

for some $s''$, respectively. From Lemma 3.1-3 we get that $s'_0 = s'_1$ and $r_1 = r'$. Since $s_1^{'} = s'$ and $\text{FV}(c_2) \subseteq v$ we get from Lemma 3.1-2 that

$$\langle c_2, s'_1 \rangle \rightarrow_{2}^{r} s'' \hspace{1cm} \text{for some } s'', \text{ and } /;/s/ gives$$

$$\langle c_1, c_2, s \rangle \rightarrow_{1}^{l} \rightarrow_{2}^{r} s''.$$ 

From $\forall P(s)$ and the assumption (1) we have that for some $s''$ and $r''$

$$\langle c_1, c_2, s \rangle \rightarrow_{1}^{l} \rightarrow_{2}^{r} s'' \hspace{1cm} \forall Q(s, s'') \text{ and } \forall R(s, r).$$

Lemma 3.1-3 gives that $r'' = r_1 + r_2 (=r)$ and thereby we get $\forall R(s, r)$ as required. This proves (2).

---

Case WHILE b DO c: Assume now that

(1) $\forall P(\text{WHILE b DO c}) \forall Q/V$

holds and we shall construct a proof of the formula in $\mathcal{R}$. As in the completeness proof for $\exists$, case WHILE b DO c, we define $P'(z)$ to be the formula

$$\exists z'. \exists x'. \exists x'. \exists x' . G(\text{WHILE b DO c}) X X X X X X X$$

where $z'$, $x'$, $x'$, $x'$ and $c'$ are as before. The relational formulas $Q'$ and $Q''$ are defined by

$$Q' \equiv G(\text{WHILE b DO c}) X_{\text{FV}(\text{WHILE b DO c})}$$

and

$$Q'' \equiv G(\text{WHILE b DO c}) X_{\text{FV}(c)} X_{\text{FV}(\text{WHILE b DO c})}.$$ 

Finally, the time formulas $R'$ and $R''$ are defined to be $E(\text{WHILE b DO c})$ and $E[c]$, respectively. A straightforward modification of the proof
of the similar result in the completeness proof for \( \mathcal{J} \), case

WHILE \( b \) DO \( c \), shows that the formula

\[
P'(z+1)\lambda b c:R'' \Rightarrow P'(z)\lambda Q'' /Vu z\]

is valid and then, by the induction hypothesis, provable in \( \mathcal{R} \). As in

the completeness proof for \( \mathcal{J} \) we also have

\[\forall P'(0) \Rightarrow \neg b,\]

\[\forall P'(z) \neg b \Rightarrow \forall v \neg Q'\]

and

\[\forall Q'' \neg Q' \Rightarrow Q'.\]

Below we shall prove that

(2) \[\forall P'(z) \neg b \Rightarrow \forall b \Rightarrow R'\]

and

(3) \[\forall E(b) \Rightarrow \forall (Q'' \Rightarrow R') \Rightarrow R'\]

so we can apply the rule /WHILE-\( \mathcal{R} /\) and get a proof of the formula

\[\exists z.P'(z) \langle \text{WHILE } b \text{ DO } c:R' \Rightarrow Q' /V.\]

As in the completeness proof for \( \mathcal{J} \), case WHILE \( b \) DO \( c \), we have

\[\forall P \Rightarrow \exists z.P'(z)\]

and

\[\forall P \Rightarrow \forall Q' \Rightarrow Q.\]

We shall prove that

(4) \[\forall P \Rightarrow \forall R' \Rightarrow R\]

so by applying first /cons-\( \mathcal{R} /\), then /inv-\( \mathcal{R} /\) and finally /cons-\( \mathcal{R} /\) we

get a proof of the required formula.

To prove (2) assume that \( \forall (P'(z) \neg b \Rightarrow \forall b \Rightarrow (s,r) \) for some pair \( (s,r) \)

of state and natural number. Then Lemma 3.2-1 gives that \( r=b^S(s) \) and

105
since \( b(s) \) holds we get from /WHILE-\$/

\[ \langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow s. \]

But then the expressiveness condition gives that \( FR'(s, r) \) and this proves (2).

To prove (3) assume that \( FE(b) \Theta R''(Q''R')(s, r) \) holds for some pair \((s, r)\) of state and natural number. Then \( r = r_1 + r_2 + r_3 \) for some \( r_1, r_2 \) and \( r_3 \) and furthermore for some \( s'\), \( FE(b)(s, r_1), FR''(s, r_2), Q''(s, s') \) and \( FR'(s, r_3) \). Lemma 3.2-1 gives that \( r_1 = b^s(s) \) and the expressiveness assumption gives that

\[ \langle c, s \rangle \rightarrow_{2} s', \text{ for some } s', \]

\[ \langle c, s \rangle \rightarrow_{2} s_1, \text{ for some } s_1 \text{ with } s_1 \in s', \text{ and furthermore } b(s) \]

and

\[ \langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow_{3} s'' \text{ for some } s''. \]

Lemma 3.1-3 gives that \( r' = r_1 \) and \( s' = s_1 \). From \( s' \in s_1 \) and \( FV(\text{WHILE } b \text{ DO } c) \subseteq V \) we get

\[ \langle \text{WHILE } b \text{ DO } c, s_0 \rangle \rightarrow_{3} s'' \]

for some \( s_0 \) using Lemma 3.1-2. Now /WHILE-\$/ gives

\[ \langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow_{1} r' + r_2 + r_3 \rightarrow s'' \]

and thereby the expressiveness assumption gives that \( FR'(s, r) \). This proves (3).

To prove (4) assume that \( FF\land R'(s, r) \) holds for some pair \((s, r)\) of state and natural number. From \( FF(s) \) and (1) we get that

\[ \langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow r', FF(s, s') \] and \( FR(s, r') \)

for some \( s' \) and \( r' \). From \( FR'(s, r) \) and the expressiveness assumption we get that
\[ \langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow s' \]

for some \( s' \). Lemma 3.1-3 gives that \( r = r' \) and thereby \( \forall R(s, r) \). This proves (4).

This completes the proof of The Completeness Theorem for \( \mathcal{R} \).

**Arithmetical Computational Models**

In Section 2.4 we proved that the proof system \( \mathcal{T} \) for total correctness is complete for every data type containing Peano Arithmetic and every arithmetical model for it. We shall now show that if, in addition, the time expressiveness condition is fulfilled then the proof system \( \mathcal{R} \) for run-time analysis is complete as well. This follows from Lemma 3.4-1. Given a data type containing Peano Arithmetic and given an arithmetical computational model for it, if the time expressiveness condition is fulfilled then so is the expressiveness condition for \( \mathcal{R} \).

**Proof:** We have to prove that for each program \( c \) we have a relational formula \( G(c) \) expressing the graph of \( c \) and a time formula \( E(c) \) expressing the exact run-time of \( c \). From Lemma 2.4-1 we get that the formulas \( G(c) \) exist. Using the time expressiveness assumption we define the formulas \( E(c) \) by structural induction on \( c \). The cases of assignment, conditional and composition are straightforward:

\[
E[x := e] \equiv E^S(e),
\]

\[
E[\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2] \equiv E(b) \oplus (b \land E(c_1) \lor \neg b \land E(c_2))
\]

and

\[
E[c_1; c_2] \equiv E(c_1) \oplus ((G(c_1) \land \lnot FV(c_2) - FV(c_1)) \cdot E(c_2)).
\]

The case of iteration is more complicated and uses the ability to
encode finite sequences of elements as a single element. More precisely, we shall encode finite parts (corresponding to FV(WHILE b DO c)) of finite sequences of states as a single vector just as in Section 2.4 but furthermore we shall encode the corresponding sequences of run-times. So define $\Theta(x, x, i, Y, y)$ to be the formula

$$
\left( \bigwedge_{j=1}^{m} \Theta_{k_j, k_j} (x_j, i, y_j) \right) \land \Theta_{\text{nat}, \text{nat}} (x, i, y)
$$

where $\text{FV}(\text{WHILE } b \text{ DO } c) = \{x_1, \ldots, x_m\}(=V)$, $x_j$ has sort $k_j$, and $\Theta_{k_j, k_j} (x_j, i, y_j)$ is as in Section 2.4. The formula $\overline{c}[\text{WHILE } b \text{ DO } c]$ is defined to be

$$
\exists n . \exists y . (\Theta(x, \text{time}, n, Y, y) \land \\
(\forall i . \forall x'. \forall \text{time}'. (i < n \land \Theta(x', \text{time}', i+1, Y, y) \land \\
\exists x''. \exists \text{time}''. (i < n \land \Theta(x'', \text{time}'', \text{time}'', \text{time}'', X') \land \\
\Theta(x'', \text{time}'', i, Y, y) \land \\
(\forall x' . \forall \text{time}'. \Theta(x', \text{time}', 0, Y, y) \Rightarrow \exists x' . \Theta(x', \text{time}', X') \land \\
\Theta(x, \text{time}, n, Y, y)) \land \\
(\forall x' . \forall \text{time}'. \Theta(x', \text{time}', 0, Y, y) \Rightarrow \exists x' . \Theta(x', \text{time}', X'))).
$$

It is straightforward (but tedious) to prove that with these definitions we have

$$
\overline{c}[\text{WHILE } b \text{ DO } c] \text{ if and only if } \langle c, s \rangle \overset{s'}{\Rightarrow} s' \text{ for some } s'
$$

for every pair $(s, r)$ of state and natural number. We omit the details.

Example 3.4-1. From Lemma 3.4-1 it follows that the data type of Extended Peano Arithmetic and its computational model (given in Example 3.1-2) satisfy the expressiveness condition for $\mathcal{R}$. Clearly, the data type contains Peano Arithmetic. The computational model is arithmetical: the encoding of a finite sequence of natural numbers as a single natural number can be accomplished by Gödel's beta function (/Sh67/). Example 3.2-1 gives that the time expressiveness
condition is fulfilled and then Lemma 3.4-1 gives that the expressiveness condition for $\mathcal{R}$ holds.

From the soundness and completeness results proved in Section 3.3 and above, respectively, it follows that the proof system $\mathcal{R}$ essentially is arithmetical sound and complete (as defined in Section 2.4) — however we have to restrict ourselves to the data types and computational models that satisfy the time expressiveness condition.

### 3.5 Example: Bubble Sorting

We now turn to a discussion of the pragmatic issues concerning the applicability of the proof system $\mathcal{R}$. As mentioned in Chapter 1 it is important that the proof system allows natural formalisations of existing informal proofs of run-time properties, because this will ensure that we can view an informal proof as a formal proof of the proof system where we have omitted some of the details (confer the transcription of Per Martin Löf's comments given in Chapter 1 /Gö83/).

We shall in this section consider the bubble sorting algorithm. We shall prove the $\mathcal{O}(n^2)$ bound on its run-time in the proof system $\mathcal{R}$ and compare this proof with the usual informal analysis. We have chosen this example partly because it is a well-known algorithm and partly because its analysis is quite simple but, on the other hand, not too simple.

Using the data type of one-dimensional arrays of Example 3.1-3 the bubble sorting algorithm can be written as follows — note the variable $l$ (of sort `array`) contains the array to be sorted.
\[ i := \text{length}(l); \]

\[ \text{WHILE } -i = 0 \text{ DO} \]

\[ (m := i; i := 0; j := 1; \]

\[ \text{WHILE } -j = m \text{ DO} \]

\[ \text{IF } l[j] > l[j+1] \text{ THEN } (x := l[j]; \]

\[ l := \text{upd}(l, j, l[j+1]); \]

\[ l := \text{upd}(l, j+1, x); \]

\[ i := j; j := j + 1) \]

\[ \text{ELSE } j := j + 1) \]

\[ \] inner

\[ \] outer

INFORMAL RUN-TIME ANALYSIS

The aim of an informal analysis of the bubble sorting algorithm will typically be to determine the order of magnitude of its worst-case time complexity. Such an analysis can be described as follows.

\text{/AHU82/}:

First consider the conditional. Each of the branches takes constant run-time and so does the test. The whole construct therefore takes constant time, that is, it is \( \Theta(1) \). The body of the inner loop is executed \( m-1 \) times, each time the run-time for both the test and the body is \( \Theta(1) \) so the total time for the inner loop is

\[(m-1) \cdot \Theta(1) = \Theta(m-1).\]

The body of the outer loop is executed at most \( \text{length}(l) \) times and the value of the variable \( i \) is decremented each time. The run-time for the body is \( \Theta(i-1) \) and for the test it is constant. The total run-time for the outer loop and thereby the complete program is

\[ (\sum_{i=1}^{\text{length}(l)} (i-1)) = \Theta(\frac{\text{length}(l)(\text{length}(l)-1)}{2}) \]

\[ = \Theta(\text{length}(l)^2). \]
We shall now see how this result can be proved in the proof system \( \mathcal{R} \). More precisely, we shall prove that there are constants \( k \) and \( k' \) such that the formula

\[
\text{TRUE} \triangleleft \text{bubble-sorting: } \text{time} \leq k \cdot \text{length}(l) \cdot \text{length}(l) + k' \triangleright \text{TRUE} / V
\]

(with \( V = \{ 1, i, j, m, x \} \)) is provable in \( \mathcal{R} \). Clearly, this means that the run-time of the algorithm is \( \Theta(\text{length}(l)^2) \). The pre- and post-conditions of the formula are chosen to be the formula TRUE as we are mainly interested in the run-time property of the program.

Before presenting the proof let us specify the two formulas that are going to be the invariants for the two while loops. We shall write \( P(z) \) for the formula \( i \leq z \) and this will be the invariant of the outer loop. For the inner loop we shall use the formula \( 0 \leq m < z + 1 \land z' + j = m \land i \leq j \) as invariant; we shall write \( P'(z') \) for this formula.

The analysis of the algorithm is presented in a bottom-up manner and has, for the sake of readability, been divided into three main parts:

- the proof for the body of the inner loop,
- the proof for the body of the outer loop, and
- the proof for the complete program.

The proof for the body of the inner loop

The body of the inner loop is a conditional. For the false branch we get, using the axiom \( \text{ass-}\mathcal{R} \), a proof of

\[
P'(z' + 1) \land (j = m) \land z < (1[j] > 1[j + 1]) \land j : = j + 1 : \text{time} = 4
\]

\[
I(\forall jz, z') - \{ j \} ^{j = j + 1} / \forall \{ z, z' \}
\]
since we, without loss of generality, can assume that $E^S(j+1)$ is
the formula $\text{time}=4$. From

$$\exists P'(z'+1) \land (j=m) \land (1[I_2 \geq z' \geq 1[I_2 + 1]) \land I(Vu[z, z']) \rightarrow
P'(z') \land maz = z \land j = j + 1$$

we get, using $/\text{inv-3}/$ and $/\text{cons-3}/$, a proof of

$$P'(z'+1) \land (j=m) \land (1[I_2 > 1[I_2 + 1]) \land \langle j: = j + 1: \text{time}=4 \rangle$$

For the true branch of the conditional we have (using $/\text{ass-3}/$)

$$P'(z'+1) \land (j=m) \land (1[I_2 > 1[I_2 + 1]) \land \langle x: = 1[I_2]: \text{time}=3 \rangle$$

$$I(Vu[z, z']) \land (l \land x = 1[I_2]/Vu[z, z'])$$

(since $E^S(1[I_2])$ can be chosen to be $\text{time}=3$),

$$\text{TRUE}(x: = \text{upd}(l, j, 1[I_2 + 1]): \text{time}=8)$$

$$I(Vu[z, z']) \land (l \land x = \text{upd}(l, j, 1[I_2 + 1]/Vu[z, z'])$$

(since $E^S(\text{upd}(l, j, 1[I_2 + 1])$ can be chosen to be $\text{time}=8$),

$$\text{TRUE}(x: = \text{upd}(l, j+1, x): \text{time}=5)$$

$$I(Vu[z, z']) \land (l \land x = \text{upd}(l, j+1, x)/Vu[z, z'])$$

(since $E^S(\text{upd}(l, j+1, x)$ can be chosen to be $\text{time}=5$),

$$\text{TRUE}(i: = j: \text{time}=2) I(Vu[z, z']) \land (i: = j/ Vu[z, z'])$$

(since $E^S(j)$ can be chosen to be $\text{time}=2$) and finally

$$\text{TRUE}(j: = j+1: \text{time}=4) I(Vu[z, z']) \land (j: = j+1/ Vu[z, z'])$$

These five proofs can be put together using the rule $/\text{-R}/$. This gives
rise to a rather complicated formula but the post-condition and the
time formula can be simplified by using first $/\text{inv-3}/$ and then $/\text{cons-3}/$
with the following formulas of the assertion language:
\[ P'(z'+1) \wedge (j=m) \wedge l[l[j]] \wedge [l[j+1]] \wedge (I(Vu{z}, z')) \wedge \{x \wedge x=1[i[j]] \}. \]

\[ (I(Vu{z}, z')) \wedge l[l[i]] \wedge a[l]=\text{upd}(l, j, l[l[j+1]]). \]

\[ (I(Vu{z}, z')) \wedge l[l[i]] \wedge a[l]=\text{upd}(l, j+1, x). \]

\[ (I(Vu{z}, z')) \wedge l[l[i]] \wedge a[l]=\text{upd}(l, j+1, x). \]

\[ \rightarrow P'(z') \wedge m=mAz=z \wedge a[j]=j+1. \]

and

\[ P'(z'+1) \wedge (j=m) \wedge l[l[j]] \wedge [l[j+1]] \wedge (I(Vu{z}, z')). \]

\[ ((\text{time}=3) \wedge (I(Vu{z}, z'))) \wedge \{x \wedge x=1[i[j]] \}. \]

\[ ((\text{time}=8) \wedge (I(Vu{z}, z'))) \wedge l[l[i]] \wedge a[l]=\text{upd}(l, j, l[l[j+1]]). \]

\[ ((\text{time}=5) \wedge (I(Vu{z}, z'))) \wedge l[l[i]] \wedge a[l]=\text{upd}(l, j+1, x). \]

\[ ((\text{time}=2) \wedge (I(Vu{z}, z'))) \wedge \{i \wedge i=j \}. \]

\[ \rightarrow \text{time}=22. \]

We thus have a proof of

\[ (2) \quad P'(z'+1) \wedge (j=m) \wedge l[l[j]] \wedge [l[j+1]] \wedge \{x \wedge x=1[i[j]] \}; \]

\[ l=\text{upd}(l, j, l[l[j+1]]); \]

\[ 1=\text{upd}(1, j+1, x); \]

\[ i=j; \]

\[ j=j+1: \text{time}=22 \rightarrow P'(z') \wedge m=mAz=z \wedge a[j]=j+1 / Vu{z}, z'. \]

The rule /IF-W can now be applied to (1) and (2) and we get a proof of the formula

\[ (3) \quad P'(z'+1) \wedge (j=m) \wedge (\text{inner: time}=29) \wedge P'(z') \wedge m=mAz=z \wedge a[j]=j+1 / Vu{z}, z'. \]

since \( l[l[j]] > l[l[j+1]] \) can be chosen to \( \text{time}=7 \) and furthermore

\[ (\text{time}=7) \wedge (\text{time}=22) \rightarrow \text{time}=29. \]

Thus we have a proof for the body of the inner loop.

The proof for the body of the outer loop

Given the proof of the formula (3) the next step is to apply the rule /WHILE-W/. We have

\[ \vdash P'(0) \rightarrow j=m, \]

\[ \vdash P'(z') \wedge (j=m) \wedge \text{Vu{z}} \rightarrow i<m \wedge m=mAz=z. \]

and

113
The invariant for the run-time of the loop is specified by the time formula \( \text{time}32^*(m-j)+3 \) and using that \( E(\tau(j=m)) \) can be chosen to be \( \text{time}=3 \) we have

\[
(4) \quad \forall \tau'(z') \forall j=m \exists \text{time}=3 \rightarrow \text{time}32^*(m-j)+3
\]

and

\[
(5) \quad \forall (\text{time}=3) \forall (\text{time}29) \forall ((m=\overline{m}z=z \wedge j=j+1) \rightarrow (\text{time}32^*(m-j)+3) \rightarrow \text{time}32^*(m-j)+3.
\]

This means that we can apply the rule \(/	ext{WHILE-2}/ \) and we get a proof of

\[
(6) \quad \exists z'. \forall \tau'(z') (\text{WHILE } \tau(j=m) \text{ DO inner: } \text{time}32^*(m-j)+3) \rightarrow (i<j=r_n z)/Vu,Iz,l.
\]

For the statement \( m:=i; i:=0; j:=1 \) it is straightforward to construct a proof of the formula

\[
P(z+1) \wedge (i=0) \exists m=i; i:=0; j:=1: \text{time}=6 \rightarrow m=\overline{m}i=0 \wedge j=1 \wedge z=\overline{z}/Vu,Iz,l.
\]

We have

\[
\forall P(z+1) \wedge (i=0) \exists m=\overline{m}i=0 \wedge j=1 \wedge z=\overline{z} \rightarrow \exists z'. P'(z') \wedge m=\overline{m}j=1 \wedge z=\overline{z}
\]

so using the rules \(/	ext{inv-2}/ \) and \(/	ext{cons-2}/ \) we get a proof of

\[
(7) \quad P(z+1) \wedge (i=0) \exists m=i; i:=0; j:=1: \text{time}=6 \exists z'. P'(z') \wedge m=\overline{m}j=1 \wedge z=\overline{z}/Vu,Iz,l.
\]

Putting (6) and (7) together with \(/	ext{-2}/ \) and using that

\[
\forall (\text{time}=6) \forall ((m=\overline{m}j=1 \wedge z=\overline{z}) \rightarrow (\text{time}32^*(m-j)+3) \rightarrow \text{time}32^*(i-1)+9
\]

and

\[
\forall P(z+1) \wedge (i=0) \wedge (m=\overline{m}j=1 \wedge z=\overline{z}) \rightarrow P(z) \wedge i<i
\]

hold we get from \(/	ext{inv-2}/ \) and \(/	ext{cons-2}/ \) that

\[
(8) \quad P(z+1) \wedge (i=0) \forall \tau: \text{time}32^*(i-1)+9 \rightarrow P(z) \wedge i<i/Vu,Iz,l.
\]

This proves a property of the body of the outer loop.
The proof for the complete program

Given the proof (8) for the body of the outer loop the next step is to apply the rule /WHILE/. We have

\[ P(0) \rightarrow i=0, \]
\[ P(z) \land i=0 \land z \rightarrow \text{TRUE}. \]

and

\[ P(i<1) \land \text{TRUE} \rightarrow \text{TRUE}. \]

For the invariant for the run-time of the loop we use the formula \( \text{time} \leq 3^2 \cdot i + 3 \). We then have

\[ P(z) \land i=0 \land \text{time} = 3 \rightarrow \text{time} \leq 3^2 \cdot i + 3 \]

and

\[ P(z) \land i=0 \land \text{time} = 3 \rightarrow \text{time} \leq 3^2 \cdot i + 3 \]

since we can choose the time formula \( E(i=0) \) to be \( \text{time} = 3 \). So applying /WHILE/ we get a proof of

\[ \exists z. P(z) \land \text{WHILE} \land i=0 \land \text{time} \leq 3^2 \cdot i + 3 \rightarrow \text{TRUE}. \]

Using /ass/ and /cons/ we now get a proof of

\[ \text{TRUE} \land i=\text{length}(l) \land \text{time} = 2 \rightarrow \exists z. P(z) \land i=\text{length}(l) \land \text{time} \leq 3^2 \cdot i + 3 \]

so using /;R/ and /cons/ with

\[ P(2) \land (i=\text{length}(l)) \land (\text{time} \leq 3^2 \cdot i + 3) \]

\[ \rightarrow \text{time} \leq 3^2 \cdot \text{length}(l) + 5 \]

we get a proof of the required formula:

\[ \text{TRUE} \land \text{bubble-sorting} \land \text{time} \leq 3^2 \cdot \text{length}(l) + 5 \rightarrow \text{TRUE}. \]
Let us now compare the two proofs, the informal one given in the beginning of this section and the formal one in the proof system given above.

To a large extent the two proofs proceed in the same way. In both analyses we have associated a run-time property with each piece of program and they are put together while we work from inside out to get a property of the run-time of the complete program. How the run-time properties actually are put together in the two analyses differ in several cases. Note for instance that the composition of statements receives a great deal of attention in the formal proof whereas it is essential ignored in the informal analysis (see for instance the proof of the formula (2)). Perhaps the most obvious difference is the way the two while loops are handled.

Consider for instance the outer loop of the program. The informal analysis follows the informal rule stated in Section 1.1 (and /AHU82/): "the run-time of a loop is the sum, over all times round the loop, of the time to execute the body and the time for evaluating the condition". The run-time of the body is \( \Theta(i-1) \), the run-time for the condition is constant and the body is executed for \( i \) being 1, 2, ..., \( \text{length}(l) \).

So for the complete loop we get the run-time

\[
\Theta(\sum_{i=1}^{\text{length}(l)} (i-1))
\]

In the formal proof the situation is different. The formula (8) gives us that the time formula \( \text{time32}^\leq(i-1)+9 \) holds for the body of the loop. In order to apply the rule /WHILE-3/ we have to find a time formula \( R \) such that
P(z)Ai=Oa\text{time}=3 \Rightarrow R, \text{ and }
\begin{aligned}
(\text{time}=3) & \otimes (\text{time}(32' (i-1)+3) \otimes ((i< i') R) \Rightarrow R
\end{aligned}
\]

hold. In the proof we have chosen R to be the formula \text{time}$\leq$32' i' i+3 and the formulas (9) and (10) show that the condition is fulfilled and we get that \text{time}$\leq$32' i' i+3 holds for the loop.

In some sense the choice of R such that (§§) holds corresponds to summing the series of (§) but the correspondance is not so close as one might wish. However, we shall here note that by restricting R to have the form \text{time}$\leq$T(i) where T is a term with free variable i we can replace (§§) by

\[
\begin{aligned}
T(0)=3, \text{ and } \\
T(i) & \leq 32' (i-1)+3)+T(i-1)
\end{aligned}
\]

where \( i< i' \Rightarrow T(i) \leq T(i') \). We then have that T(i) corresponds to the sum of the first i elements of the series (§); in fact, finding a solution to (§§§) using the technique of summing series (/AHU82/) corresponds very closely to calculating the sum in (§). This will be discussed further in Chapter 6.

3.6 ALTERNATIVE APPROACHES

The proof system \( \mathcal{R} \) imposes a certain style on the proofs of run-time properties. The overall impression from the worked example of the previous section is that to a large extent this style agrees with the one of the traditional informal analyses. An obvious question is how it compares with the alternative (logical) approaches reviewed in Section 1.2. We shall in this section try to answer this question by comparing both the "formal power" and the "styles of proofs" of the approaches. Furthermore, we shall consider an interesting variant of
Shaw's method (/Sh79/) where the "flow of run-time information" is backwards rather than forwards as in Shaw's approach.

**THE METHOD OF PROGRAMMED COUNTERS**

As mentioned in Section 1.2 the idea in this approach is to insert statements in the program that update the time used so far and then to use the proof system for total correctness to prove properties of these counter variables and thereby the run-time of the original program. We shall now give a more formal description of the method and afterwards compare it with the proof system $\mathcal{R}$.

In order to perform the transformation of programs we must require that we have terms (rather than time formulas) in the assertion language for the run-time requirements of the basic operations of the data type. So we shall impose the following **exact time term expressiveness condition** on the data type and its computational model:

- For each sort $k$ of the data type there is a term $U^+(x)$ with the free variable $x$ of sort $k$ and such that for every state $s$
  
  $$U^+(x)(s) = x(s)^+,$$

- For each function symbol $f$ of arity $(k_1 \ldots k_m, k)$ there is a term $U_f(x_1, \ldots, x_m)$ with free variables $x_1, \ldots, x_m$ of sorts $k_1, \ldots, k_m$, respectively, and such that for every state $s$
  
  $$U_f(x_1, \ldots, x_m)(s) = f^+(x_1(s), \ldots, x_m(s)).$$

- For each relation symbol $p$ of arity $k_1 \ldots k_m$ there is a term $U_p(x_1, \ldots, x_m)$ with free variables $x_1, \ldots, x_m$ of sorts $k_1, \ldots, k_m$, respectively, and such that for every state $s$
  
  $$U_p(x_1, \ldots, x_m)(s) = p^+(x_1(s), \ldots, x_m(s)).$$
Note that if this condition is fulfilled then so is the time expressiveness condition of Section 3.2.

Assume now that the exact time term expressiveness condition is fulfilled. Then we have terms $U(e)$ and $U(b)$ for the time required to evaluate the term $e$ and the boolean expression $b$, respectively. They can be defined as follows:

<table>
<thead>
<tr>
<th>$e$</th>
<th>$U(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$U^+(x)$</td>
</tr>
<tr>
<td>$v$</td>
<td>$U^+(v)$</td>
</tr>
<tr>
<td>$f(e_1,\ldots,e_m)$</td>
<td>$U(e_1)+\ldots+U(e_m)+U_f(e_1,\ldots,e_m)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$b$</th>
<th>$U(b)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE (resp. FALSE)</td>
<td>0</td>
</tr>
<tr>
<td>$p(e_1,\ldots,e_m)$</td>
<td>$U(e_1)+\ldots+U(e_m)+U_p(e_1,\ldots,e_m)$</td>
</tr>
<tr>
<td>$\neg b$</td>
<td>$U(b)$</td>
</tr>
<tr>
<td>$b_1 \land b_2$</td>
<td>$U(b_1)+U(b_2)$</td>
</tr>
<tr>
<td>$b_1 \lor b_2$</td>
<td>$U(b_1)+U(b_2)$</td>
</tr>
<tr>
<td>$b_1 \rightarrow b_2$</td>
<td>$U(b_1)+U(b_2)$</td>
</tr>
</tbody>
</table>

Furthermore, define $U^S(e)$ to be $U(e)+U^+(e)$. Corresponding to Lemma 3.2-1 we then have

**Lemma 3.6-1.** If the exact time term expressiveness condition holds for a data type and its computational model then for every term $e$, boolean expression $b$ and state $s$

- $U(e)(s) = e^S(s)$,
- $U^S(e)(s) = e^S(s) + e(s)^+$,
- $U(b)(s) = b^S(s)$.

The transformation of programs can now be specified as follows (for
the sake of simplicity we shall here assume that the variable time is a program variable (meaning that it can be used in programs) but that it does not occur in any of the original programs):

<table>
<thead>
<tr>
<th>c</th>
<th>c'</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:=e</td>
<td>x:=e; time:=time+U^+(e)</td>
</tr>
<tr>
<td>IF b THEN c₁ ELSE c₂</td>
<td>IF b THEN (time:=time+U(b);c₁⁺) ELSE (time:=time+U(b);c₂⁺)</td>
</tr>
<tr>
<td>c₁;c₂</td>
<td>c₁⁺;c₂⁺</td>
</tr>
<tr>
<td>WHILE b DO c</td>
<td>WHILE b DO (time:=time+U(b);c⁺); time:=time+U(b)</td>
</tr>
</tbody>
</table>

It is straightforward to prove (by induction on the formal proofs in \( \mathcal{F} \)) that

\[
\begin{align*}
\langle c, s \rangle & \xrightarrow{\mathcal{F}} s', \\
\langle c', s_0 \rangle & \xrightarrow{\mathcal{T}} s_{time}^{r'} & \text{for some } r'.
\end{align*}
\]

The method of programmed counters can now be described as follows:

1. transform the given program \( c \) into the program \( c' \)
2. construct a proof of some formula of the form \( P_{time=0} \langle c' \rangle Q/V \) in \( \mathcal{F} \) (and with \( time \notin \text{FV}(Q) \))
3. conclude that if \( P \) holds for the initial state of \( c \) then \( c \) terminates and \( Q \) holds for the run-time of \( c \) (and the initial and the final state).

The "soundness" of the method follows from the soundness of the proof system \( \mathcal{F} \) and the relationship given by (§).

Let us now compare this approach with that of the proof system \( \mathcal{R} \).

It can be proved that if the exact time term expressiveness condition...
is fulfilled then anything provable about the run-time of a program in $\mathcal{R}$ will also be provable using the method of programmed counters. Assume namely that we have a proof of the formula $P(c:R\triangleright Q/V)$ in $\mathcal{R}$. The Soundness Theorem for $\mathcal{R}$ (in Section 3.3) then gives that the formula is valid and using (§) it can be proved that the formula $P_{\text{time}=0}\langle c^{+}\triangleright Q_{R}/Vu_{\text{time}}\rangle$ is valid (where $R$ is the formula obtained by replacing all free variables except $\text{time}$ by the corresponding shadow variable). The Completeness Theorem for $\mathcal{T}$ (in Section 2.4) then gives that the formula is provable in $\mathcal{T}$ (assuming that the expressiveness condition for $\mathcal{T}$ is fulfilled). So if the property expressed by the formula $P(c:R\triangleright Q/V)$ is provable in $\mathcal{R}$ then the corresponding property $Q_{\bar{R}}$ for the run-time of $c$ is provable using the method of programmed counters.

In general, it is not the case that any property proved about the run-time of a program using the method of programmed counters can be proved in $\mathcal{R}$. The reason is simply that much more properties can be expressed (and proved) in the former method, for instance properties relating the run-time to the final values of the program variables rather than the initial ones. However, note that if we restrict our attention to formulas of the form $P_{\text{time}=0}\langle c^{+}\triangleright Q_{R}/Vu_{\text{time}}\rangle$ where $\text{FV}(Q)\subseteq V\bar{u}$ and $\text{FV}(R)\subseteq V\{\text{time}\}$ then provability of this formula in $\mathcal{T}$ will imply provability of the formula $P(c:R\triangleright Q/V)$ in $\mathcal{R}$ - the arguments are similar to those given above for the inverse implication.

The conclusion is therefore that the method of the programmed counters is theoretically more powerful than the proof system $\mathcal{R}$ but the conjecture is that in practice it is not - the reason being that we really only are interested in relationships between the initial states and the corresponding run-times. The completeness result for
the proof system $\mathcal{R}$ in Section 3.4 suggests that $\mathcal{R}$ is sufficiently powerful to prove that sort of properties and thereby that the extra strength of the method of programmed counters is not really necessary.

As we already noted in Section 1.3 the method of programmed counters gives rise to run-time analyses that are rather different from the traditional informal analyses because of the program transformation that has to be performed. So from this more pragmatic point of view it is quite obvious that we should prefer the styles of proofs imposed by the proof system $\mathcal{R}$.

**Shaw's Method**

The idea in Shaw's method is to count the time used so far in a pseudo variable time and then update its value indirectly by manipulating the time formulas (see Section 1.2, /Sh79/). In order to compare the method with that of the proof system $\mathcal{R}$ we shall first reformulate it in our framework, that is, present it as a Hoare-like proof system.

As mentioned in Section 1.2 Shaw's method is based on predicate transformers. Given a program $c$ and a time formula $R'$ there is another time formula $cp(c,R')$ called the cost-precondition of $c$ and $R'$. Corresponding to this we shall extend the formulas of the total correctness proof system $\mathcal{T}$ with a new pair of pre- and post-conditions for expressing run-time properties. The new formulas will be written $P&R(c)Q&R'/V$ where $P$, $c$, $Q$ and $V$ are as before and $R$ and $R'$ are time formulas. Intuitively, $R$ will correspond to the cost-precondition of $c$ and $R'$ (more precisely, $R\Rightarrow cp(c,R')$ will hold). Remember that $R'$ is a property of the time spent until the execution of $c$ is completed whereas $R$ is a
property of the time used before the execution of c starts. This is illustrated on the figure below:

We shall construct a proof system, called $\mathcal{T}$, with formulas of the form $P \& R(c) Q \& R'/V$. A well-formedness condition is imposed requiring that

- $FV(P) \subseteq V$, $FV(c) \subseteq V$, $FV(Q) \subseteq V \cup \mathit{time}$, and
- $FV(R) \subseteq V \cup \{\mathit{time}\}$, $FV(R') \subseteq V \cup \{\mathit{time}\}$.

The validity of the formula, denoted $\mathcal{F}P \& R(c) Q \& R'/V$ is formally defined by

for every state $s$, if $\mathcal{F}P(s)$ holds then for some state $s'$ and natural number $r$

$$<c,s> \xrightarrow{s'} s', \mathcal{F}Q(s,s')$$

and for every natural number $r'$, $\mathcal{F}R(s,r')$

implies $\mathcal{F}R'(s',r'+r)$.

Assuming that the exact time term expressiveness condition of the previous subsection is fulfilled the axioms and rules of the proof system $\mathcal{T}$ are as in the table below. We shall write $\mathcal{F}P \& R(c) Q \& R'/V$ if the formula is provable using the axioms and rules of $\mathcal{T}$ (with the usual restriction on their applicability, see for instance Section 3.2).

The idea in the axioms and rules of $\mathcal{T}$ is that the time formulas are transformed backwards through the programming constructs and thus
The proof system $\mathcal{T}$

\[\text{/ass-}\mathcal{T}/ \quad \frac{P \text{e}^x \text{time} + U^e(e) \langle x := e \rangle_{V - \{x\} \wedge x = e} \text{time}}{P \text{e}^x \text{time}}\]

\[\text{/IF-}\mathcal{T}/ \quad \frac{P \text{a}^b \text{time} + U^b(b) \langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \rangle_{Q \wedge R'} \text{time}}{P \text{a}^b \text{time} + U^b(b) \langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 \rangle_{Q \wedge R'}}\]

\[\text{/i-}\mathcal{T}/ \quad \frac{P \text{r}^c \text{time} \langle c_1; c_2 \rangle_{Q_1 \wedge Q_2 \wedge R'} \text{time}}{P \text{r}^c \text{time} \langle c_1; c_2 \rangle_{Q_1 \wedge Q_2 \wedge R'}}\]

\[\text{/WHILE-}\mathcal{T}/ \quad \frac{\exists z. P(z) \text{time} + U^b(b) \langle \text{WHILE } b \text{ DO } c \rangle_{Q \wedge R'} \text{time}}{\exists z. P(z) \text{time} + U^b(b) \langle \text{WHILE } b \text{ DO } c \rangle_{Q \wedge R'}}\]

where $z$ is a variable of sort $\text{nat}$ satisfying $z \notin V$

\[\text{/cons-}\mathcal{T}/ \quad \frac{P \rightarrow P', P \text{a} \rightarrow R', P \text{r} \langle c \rangle_{Q' \wedge R'1 \wedge R'} \text{time}}{P \text{r} \langle c \rangle_{Q' \wedge R'}}\]

\[\text{/inv-}\mathcal{T}/ \quad \frac{P \text{a}^c \langle c \rangle_{Q \wedge R'} \text{time}}{P \text{a} \text{a}^c \langle c \rangle_{Q \wedge R'}}\]

reflecting that the run-time information goes forwards - exactly as it happens in the weakest preconditions given in Section 1.2. Only the rule /WHILE-\mathcal{T}/ is different because we in a Hoare-like proof system must choose a formula of the assertion language to replace the infinite disjunction of the weakest precondition. The idea in /WHILE-\mathcal{T}/ is that the time formula $R$ is a property of the run-time required from the beginning of the computation and until a test on $b$ in the loop has been completed. This is illustrated on the following figure.
The proof system $\mathcal{F}$ can be proved to be sound in the following sense:

**The Soundness Theorem for $\mathcal{F}$**

Given a data type and a numerical computational model for it, if the exact time term expressiveness condition is fulfilled then for every well-formed formula $P \& R(c) \land Q \land R'/V$ of $\mathcal{F}$

$$\mathcal{F} + P \& R(c) \land Q \land R'/V \implies F P \& R(c) \land Q \land R'/V.$$

The proof of this result is a straightforward modification of that proving the soundness result for the proof system $\mathcal{T}$ in Section 2.3 so we omit the details here.

We can also obtain a completeness result for $\mathcal{F}$. The expressiveness condition for $\mathcal{F}$ is slightly different from that for $\mathcal{R}$ in that it requires the existence of some weakest preconditions for the time formulas. More precisely, the expressiveness condition for $\mathcal{F}$ is fulfilled if

- the exact time term expressiveness condition is fulfilled,
- the expressiveness condition for $\mathcal{F}$ is fulfilled,
- for every while program $c$ and time formula $R$ there is a time formula $WP[c,R]$ with $FV(WP[c,R]) \subseteq FV(c) \cup FV(R)$ and satisfying that for every pair $(s,r)$ of state and natural number

$$WP[c,R](s,r)$$
if and only if
\[ \langle c,s \rangle \rightarrow s' \text{ and } \forall R(s',r+r') \] for some \( s' \) and \( r' \).

Using this expressiveness condition we can prove (see Appendix A):

The Completeness Theorem for \( \mathcal{F} \)

Given a data type and a numerical computational model for it,
if the expressiveness condition for \( \mathcal{F} \) is fulfilled then for every formula \( P \& R \langle c \rangle \forall Q \& R'/\forall \) of \( \mathcal{F} \)
\[ \forall P \& R \langle c \rangle \forall Q \& R'/\forall \text{ implies } \mathcal{F} \vdash P \& R \langle c \rangle \forall Q \& R'/\forall \text{.} \]

Let us now compare the formal power of the two proof systems \( \mathcal{R} \) and \( \mathcal{F} \). We shall restrict our attention to data types containing Peano Arithmetic and their arithmetical computational models and furthermore we shall assume that the exact time term expressiveness condition is fulfilled. We can then prove that \( \mathcal{F} \) is at least as powerful as \( \mathcal{R} \). To see this we shall first note that the expressiveness condition for \( \mathcal{F} \) is fulfilled for the data types and computational models we consider (see Appendix A). Assume now that the formula \( R \langle c \rangle \forall Q \& V \) is provable in \( \mathcal{R} \). Then it is valid according to The Soundness Theorem for \( \mathcal{R} \) and it is easy to verify that then the formula \( P \land X = X' \land \text{time} = \langle \forall Q \& R_x \rangle X' / \forall V \) (where \( X' \) is a vector of distinct new variables of the same length as \( X \) and of the appropriate sorts and \( V' \) is the set of these variables) is valid. But then The Completeness Theorem for \( \mathcal{F} \) gives that the formula is provable in \( \mathcal{F} \).

Obviously, it is not the case that \( \mathcal{R} \) and \( \mathcal{F} \) are equally powerful since we can express run-time properties in the formulas of \( \mathcal{F} \) that cannot be expressed in the formulas of \( \mathcal{R} \). As we argued for the method of programmed counters in the previous subsection this is not so important because we can express all the run-time properties we really
want to consider in the formulas of $\mathcal{R}$.

In order to compare the style of proofs imposed by the two proof systems we shall consider the bubble sorting algorithm of Section 3.5 once again. When using the proof system $\mathcal{T}$ one is invited to look for time formulas expressing properties of the time used so far - this is simply the idea behind the proof system. So for instance when analysing the true-branch of the conditional of the bubble sorting algorithm we will look for a time formula for the run-time required to

- execute the initial statements of the program,
- execute the body of the outer loop a number of times,
- execute the body of the inner loop a number of times, and
- evaluate the test of the conditional.

In the informal analysis one is not at all interested in the sum of these time requirements and from this it should be clear that the style of proof imposed by the proof system $\mathcal{R}$ should be preferred over that suggested by $\mathcal{T}$. Furthermore the proof system $\mathcal{T}$ does not force one to prove run-time properties that relate the run-time to the initial values of the program variables. Since we are only interested in that sort of properties of the run-time it will from a pragmatical point of view be regarded as a weakness of the proof system $\mathcal{T}$. So the conclusion is that the proof system $\mathcal{R}$ should be preferred over $\mathcal{T}$.

A variant of Shaw's method

The idea in Shaw's method is to keep an implicit count of the time used so far. An alternative would be to keep track of the time needed for the rest of the computation. As we shall see below this gives rise to a proof system that in some respects are very similar to that of
the previous subsection but there are also important differences, for instance, one will be forced to prove run-time properties relating the initial values of the program variables to the corresponding run-time.

We shall now construct a proof system called $\mathbf{3}$ based on the idea of keeping track of the time required for "the rest of the computation". The formulas of this proof system will have the form $P \land R < c \land Q \land R' / V$ as those of $\mathbf{3}$ above but the interpretation is different. Here $R'$ is a property of the time required for the part of the program following $c$ and $R$ is a property of the time required to execute both $c$ and what follows $c$. This is illustrated on the following figure:

Formally the validity of the formula $P \land R < c \land Q \land R' / V$ (ambiguously written $FP \land R < c \land Q \land R' / V$) is defined as follows

for every state $s$, if $FP(s)$ holds then for some state $s'$ and natural number $r$

$\langle c, s \rangle \xrightarrow{s'} s'$, $FQ(s, s')$ and for every natural number $r'$, $FR'(s', r')$

implies $FR(s, r+r')$.

Note that the "flow of run-time information" in the formula is backwards (that is, from right to left) in contrary to what is the case for the formulas of $\mathbf{3}$. Furthermore, note that $FP < c : R \land Q / V$ holds if and only if $FP \land R < c \land Q \land time=0 / V$ holds.
Assuming that the time expressiveness condition of Section 3.2 is fulfilled we have the following axioms and rules in the proof system $\mathcal{B}$:

The proof system $\mathcal{B}$

\[
\begin{align*}
\text{/ass-} & \quad P \land\exists x : e \rightarrow t \cdot \forall x : e \rightarrow R
\
\text{/IF-} & \quad P \land\exists R \{ c_1 > Q \lor R' \}, P \land\exists R \{ c_2 > Q \lor R' \}
\
\text{/;=} & \quad P \land\exists R \{ c_1 \land Q \land R'' \}, P \land\exists R \{ c_2 \land Q \land R'' \}
\
\text{/WHILE-} & \quad \exists z . P(z) \land\exists E(b) \rightarrow Q, Q' \rightarrow Q, P(z) \land\exists R' \rightarrow R
\
\text{/cons-} & \quad P \rightarrow P', R \rightarrow R', P \land\exists R \{ c \land Q' \lor R' \}, Q' \rightarrow Q, R' \rightarrow R'
\
\text{/inv-} & \quad P \land\exists R \{ c \land Q \land R' \}
\
\end{align*}
\]

We shall write $\mathcal{B} \vdash P \land\exists R \{ c \land Q \land R' \}$ for provability of the formula $P \land\exists R \{ c \land Q \land R' \}$ using the axioms and rules above (and with the usual restrictions on their applicability).

Let us now explain the idea behind these axioms and rules. In the axiom /ass- $\mathcal{B}$/ $R$ is a property of the run-time of the program that follows the assignment statement but relative to its final state. Therefore $R^e_x$ is a property of the same run-time but relative to the
initial state and by "adding" the run-time property $E^S(e)$ for the assignment statement itself we get the required run-time property.

In /IF-/ we assume that the run-time properties for the two branches have been unified just as in the rule /IF-/ considered earlier. The rule /;-/ is straightforward: the information about the run-time for "the rest of the computation" is updated by "pulling it backwards" through the statements, one at the time. This rule (as well as /;-/ above) thus avoids the explicit transformation of time formulas that happens in the rule /;-. A similar simplification has happened in the rule /WHILE-/. The time formula $E(b)@R$ is an invariant for the run-time of a number of executions of the body of the loop. If the body is not executed at all (the test evaluates to false) then the hypothesis $P(z)\land \neg b \Rightarrow R \Rightarrow R$ ensures that the total run-time satisfies $E(b)@R$. In the case where the body is executed it will also hold because of the hypothesis about the body. The idea is further illustrated on the figure:

The proof system $\mathcal{B}$ can be proved to be sound in the same sense as the proof system $\mathcal{R}$.
The Soundness Theorem for $\mathfrak{B}$

Given a data type and a numerical computational model for it, if the time expressiveness condition is fulfilled then for every well-formed formula $P \land R \rightarrow Q \land R'/V$

$$\mathfrak{B} \vdash P \land R \rightarrow Q \land R'/V \text{ implies } \exists P \land R \rightarrow Q \land R'/V.$$  

The proof of this result is a straightforward modification of that for the soundness result of $\mathfrak{T}$ in Section 2.3 so we omit the details here.

As a consequence of the soundness result for $\mathfrak{B}$ we get that if a formula of the form $P \land R \rightarrow Q \land \text{time}=0/V$ is provable in $\mathfrak{B}$ then the corresponding property $P \rightarrow Q/V$ is provable in $\mathfrak{R}$. To see this observe that The Soundness Theorem for $\mathfrak{B}$ gives that the formula $P \land R \rightarrow Q \land \text{time}=0/V$ is valid and thereby that $P \rightarrow Q/V$ is valid. The Completeness Theorem for $\mathfrak{R}$ then gives that this formula is provable in $\mathfrak{R}$ (assuming that the expressiveness condition for $\mathfrak{R}$ is fulfilled). We also have the following result that allows us to conclude that if the formula $P \rightarrow Q/V$ is provable in $\mathfrak{R}$ then the formula $P \land R \rightarrow Q \land \text{time}=0/V$ is provable in $\mathfrak{B}$:

The Completeness Theorem for $\mathfrak{B}$

Given a data type and a numerical computational model for it, if the expressiveness condition for $\mathfrak{R}$ is fulfilled then for every well-formed formula $P \land R \rightarrow Q \land \text{time}=0/V$

$$\exists P \land R \rightarrow Q \land \text{time}=0/V \text{ implies } \mathfrak{B} \vdash P \land R \rightarrow Q \land \text{time}=0/V.$$  

We shall prove this result by showing how to transform a proof of the formula $P \rightarrow Q/V$ in $\mathfrak{R}$ into a proof of the corresponding formula $P \land R \rightarrow Q \land \text{time}=0/V$ in $\mathfrak{B}$. Since the proof system $\mathfrak{R}$ is complete (see Section 3.4) this is sufficient to prove the completeness result for $\mathfrak{B}$.  

131
More precisely we shall now show how to transform a proof of the formula \( P(c:R>Q/V) \) in \( \mathcal{R} \) into a proof of the formula \( P&R@Q&R'/V \) in \( \mathcal{B} \) for any time formula \( R' \). Depending on the last axiom or rule applied in the proof in \( \mathcal{R} \) we proceed as follows:

**Case \( \text{ass-R} \):** By assumption we have a proof

\[
\mathcal{R} \vdash P(x := e : E(e)) \leftarrow \mathcal{V} - \{x\} \wedge x = e/V.
\]

From \( \text{ass-R} \) we obtain a proof

\[
\mathcal{B} \vdash P&E S(e) \Theta R' x \leftarrow e : E(x) \leftarrow \mathcal{V} - \{x\} \wedge x = e & R'/V
\]

and using \( \text{cons-B} \) with \( FE S(e) \Theta R' e \rightarrow E_S(e) \Theta (\mathcal{V} - \{x\} \wedge x = e) \rightarrow R' \) we get

\[
\mathcal{B} \vdash P&E S(e) \Theta (\mathcal{V} - \{x\} \wedge x = e) \rightarrow R' \leftarrow e : E(x) \leftarrow \mathcal{V} - \{x\} \wedge x = e & R'/V
\]

as required.

**Case \( \text{IF-R} \):** Assume now that we have obtained the proof

\[
\mathcal{R} \vdash P(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 : E(b) \Theta R>Q/V
\]

by applying \( \text{IF-R} \) to the proofs

\[
\mathcal{R} \vdash Pab \leftarrow c_1 : Q/V
\]

and

\[
\mathcal{R} \vdash Pab \leftarrow c_2 : Q/V.
\]

The induction hypothesis gives that the latter proofs can be transformed into proofs

\[
\mathcal{B} \vdash Pab & R@Q&R'/V
\]

and

\[
\mathcal{B} \vdash Pab & R@Q&R'/V.
\]

Using \( \text{IF-B} \) we then get the required proof

\[
\mathcal{B} \vdash P&E (b) \Theta R@Q&R'/V.
\]
Case /;/:-/: We have now obtained the proof
\[ \exists c_1, c_2 : R_1 \otimes (Q_1 \otimes R_2) \Rightarrow Q_1 \Rightarrow Q_2 /\forall \]
from the proofs
\[ \exists c_1 : R_1 \otimes P \Rightarrow Q_1 /\forall \]
and
\[ \exists c_2 : R_2 \Rightarrow Q_2 /\forall . \]
Using the induction hypothesis these two proofs can be transformed into proofs
\[ \exists \bar{P} \& R_1 \otimes (P' \& Q_1, (R_2 \& Q_2, R')) < c_1, P' \& Q_1, R_2 \& Q_2, R' /\forall \]
and
\[ \exists \bar{P} \& R_2 \& Q_2, R' < c_2, Q_2 \& R' /\forall . \]
So using first /;/:-/ and then /cons-\( \exists /\) with
\[ \exists \bar{R}_1 \otimes ((P' \& Q_1) \Rightarrow (R_2 \& Q_2, R')) \Rightarrow R_1 \otimes Q_1, R_2 \& (Q_1 \& Q_2), R' \]
we get the required proof
\[ \exists \bar{P} \& R_1 \otimes Q_1, R_2 \& (Q_1 \& Q_2), R' < c_1, c_2, Q_2 \& R' /\forall . \] ///

Case /;WHILE-/: We have now obtained the proof
\[ \exists c, R \Rightarrow P(z) \Rightarrow \text{WHILE } b \text{ DO } c : R /\forall \]
from the proof
\[ (1) \exists c, P(z+1) \Rightarrow c : R_0 \Rightarrow P(z) \& Q' /\forall \{z\} \]
together with the facts \( \forall P(0) \Rightarrow b, \forall P(z) \Rightarrow (b \& \text{L}_V) \Rightarrow Q, \forall Q' \Rightarrow Q, \)
\( \forall P(z) \Rightarrow b \& \text{E}(b) \Rightarrow R \) and \( \forall E(b) \Rightarrow Q' \Rightarrow (R \& Q' \Rightarrow R). \) Now define \( R_1 \) to be the time formula \( (P(z) \& b \Rightarrow R') \Rightarrow (R_0 \& Q' \Rightarrow (R \& Q' \Rightarrow R')). \) The induction hypothesis gives that the proof \((1)\) can be transformed into a proof
\[ \exists \bar{P}(z+1) \& b \& R_0 \& (P(z) \& Q') \Rightarrow (E(b) \& R_1) \Rightarrow c : P(z) \& Q' \Rightarrow E(b) \& R_1 /\forall \{z\} \].
It can be proved that \( \exists R_1 \Rightarrow (P(z) \& Q') \Rightarrow (E(b) \& R_1) \Rightarrow R_1 \) so using /cons-\( \exists /\)
we get the proof
\[ \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \].
Since \[ \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \] holds we can apply /WHILE-\[ \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \] and get
\[ \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \). \]
It can be proved that \[ \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \] so using /cons-\[ \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \] we get the required proof
\[ \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \].
Case /cons-\[ \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \]: Straightforward and therefore omitted.
Case /inv-\[ \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \]: Straightforward and therefore omitted.
This completes the specification of the transformation of a proof in \( R \) into one in \( B \) and thereby The Completeness Theorem for \( B \) has been proved.

This shows that the proof system \( B \) is at least as powerful as \( R \). As mentioned earlier we get from the soundness result for \( B \) and the completeness result for \( R \) that if some formula \[ \neg \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \] is provable in \( B \) then the corresponding formula \[ \neg \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \] is provable in \( R \). It is not clear whether this result also can be proved directly by transformation of proofs as above. To illustrate the kind of problems that arise let us consider the general case where we have a proof
\[ \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \) and want a proof of the formula \[ \exists z \forall z (\neg b \land R \rightarrow R_1 \land \neg R \lor \neg z \} \] in \( R \) where \( R' \) is some formula constructed from \( R \) and \( R' \).

A first suggestion might be to define \( R' \) such that \[ \forall R'(s,r) \] holds if and only if for every state \( s' \) and natural number \( r' \), \[ \forall Q(s,s') \) and \[ \forall R'(s',r') \] implies \[ \forall R(s,r+r') \]. However, as the following example shows this will not work. Assume that we have a data type and a computational
model for it where the program \( x := x/2 \) takes time \( x/2 \). A proof of the (valid) formula \( \text{TRUE} & \text{time} = x(x := x/2) \text{TRUE} & \text{time} = x/V \) in \( \mathfrak{B} \) should then give rise to a proof of the formula \( \text{TRUE} < x := x/2; \text{FALSE} > \text{TRUE} / V \) in \( \mathcal{A} \) since the formula constructed from the two (identical) time formulas \( \text{time} = x \) and the relational formula \( \text{TRUE} \) must be equivalent to \( \text{FALSE} \). But the formula above is not valid so we cannot have a proof of it in \( \mathcal{A} \).

The formula \( Q \) thus seems to be too weak when combining the two formulas \( R \) and \( R' \) into \( R'' \). So the next suggestion might be to use the strongest possible formula, namely \( \mathcal{FAG}[c]_V \mathcal{A} \mathcal{I}_{V-FV}(c) \), and define \( R'' \) such that \( \mathcal{F} R''(s, r) \) holds if and only if for every state \( s' \) and natural number \( r' \), \( \mathcal{F} \mathcal{A} \mathcal{G}[c]_V \mathcal{A} \mathcal{I}_{V-FV}(c)(s, s') \) and \( \mathcal{F} R'(s', r') \) implies \( \mathcal{F} R(s, r + r') \). This would for instance ensure that we have a proof of the formula \( \text{TRUE} < x := x/2; \text{time} = x/2 > \text{TRUE} / V \) in \( \mathcal{A} \). But still the transformation of proofs does not work. To see why, consider the composition rule and assume that we from the proofs

\[
\begin{align*}
\mathfrak{B} & \vdash \text{TRUE} & \text{time} = x < x := x/2 > \text{TRUE} & \text{time} = x/V \\
\text{and (for some program } c) \\
\mathfrak{B} & \vdash \text{TRUE} & \text{time} = x < c > \text{TRUE} & \text{time} = 0/V
\end{align*}
\]

have obtained a proof

\[
\begin{align*}
\mathfrak{B} & \vdash \text{TRUE} & \text{time} = x < x := x/2; c > \text{TRUE} & \text{time} = 0/V
\end{align*}
\]

using \( /; \mathfrak{B} \). The idea will then be that we from the transformed proofs

\[
\begin{align*}
\mathcal{A} & \vdash \text{TRUE} < x := x/2; \text{time} = x/2 > \text{TRUE} / V \\
\text{and}
\end{align*}
\]

\[
\begin{align*}
\mathcal{A} & \vdash \text{TRUE} < c; \text{time} = x > \text{TRUE} / V
\end{align*}
\]

should obtain a proof of the formula

\[
\text{TRUE} < x := x/2; c; \text{time} = x > \text{TRUE} / V.
\]
But this is not possible since we cannot deduce that $\text{time}=x$ holds from $(\text{time}=x/2) \& \text{TRUE}$ ($\text{time}=x$). So we shall also need stronger post-conditions in the formulas. However, if we are going to change the post-conditions of the formulas we can hardly say that we give a direct transformation of the proofs in $\mathcal{B}$ into proofs in $\mathcal{R}$.

In order to compare the styles of proofs imposed by the two proof systems $\mathcal{B}$ and $\mathcal{R}$ we shall once again consider the bubble sorting program. We shall discuss the choice of time formula that should hold just before the execution of the true branch of the conditional of the program is started. The idea behind the proof system $\mathcal{B}$ is that the time formulas express properties for the run-time of the rest of the computation. In our case this is the time required in order to

- execute the true branch of the conditional,
- execute the body of the inner loop a number of times, and
- execute the body of the outer loop a number of times.

In the informal analysis (see Section 3.5) we do not at all consider the run-time for this part of the execution of the program so from a pragmatic point of view we shall prefer the proof system $\mathcal{R}$ over $\mathcal{B}$. The proof system $\mathcal{B}$ seems not to give rise to natural formalisations of the existing informal analyses.

3.7 CONCLUDING REMARKS

We have in this chapter shown how the proof system for total correctness presented in Chapter 2 can be extended in various ways to prove run-time properties of while programs. We have mainly considered an approach based on the idea that in order to prove a property of the
run-time of a composite program we must first prove some properties of the run-time of its constituents and then this information is put together to obtain a property of the total run-time for the program.

The knowledge of the run-time requirements of the basic operations of the language is given by a computational model for the data type. The computational models, as introduced in Section 3.1, are slightly more general than the algebras with norms and resource charge functions introduced by Asveld and Tucker in /AsTu82/. Their idea is to introduce a norm on the algebra (essentially a model in our terminology) being a mapping from the set of data elements to the set of natural numbers. The resource charge functions correspond to our cost-functions except that they are defined from some numerical functions giving the resource requirements for the operations in terms of the norm of their arguments rather than the arguments themselves. For practical purposes this seems quite realistic and the development of the previous sections can easily be modified to use such a notion of computational model.

Intuitively, the idea is that a model for a data type specifies an implementation of it and a computational model will furthermore specify the run-time requirements of the various operations. However, it is well-known that not every model is computable, that is, is a realistic specification of an implementation of the data type. Furthermore, a computational model with an underlying computable model need not be a realistic specification of an implementation - there is, for instance, no requirements ensuring that axioms corresponding to Blum's axioms for a complexity measure are fulfilled (see /HoU179/). Asveld and Tucker discuss a related topic in /AsTu82/ where they introduce the polynomial time implementable data types. However more research is
needed and especially, it will be interesting to investigate the fulfillment of the expressiveness conditions for these realistic specifications of implementations of data types. Work in this area has already been presented by Bergstra and Tucker in /BeTu80/ where they consider the expressibility of the strongest post-conditions in the case of computable models.

The proof in $\mathcal{R}$ formalising the informal analysis of the bubble sorting algorithm in Section 3.5 is approximately five times as long as the informal analysis (but, of course, almost any factor can be obtained by including or omitting details in the proof). In itself it is not surprising that the formal proof is longer than the informal one, a similar effect can for instance be observed for correctness proofs. What may be interesting to know is whether an extra factor is added because we are proving run-time properties and not just total correctness. Measured by the number of applications of axioms and rules in the two proof systems $\mathcal{R}$ and $\mathcal{T}$, a proof of $P(\mathbf{c}: \mathcal{R})Q/V$ in $\mathcal{R}$ and one of $P(\mathbf{c})Q/V$ in $\mathcal{T}$ will have approximately the same length. There may be situations where we will apply one of the general rules of $\mathcal{R}$ but not the corresponding one in $\mathcal{T}$. One reason for this is that we may perform extra deductions about the run-time properties but it may also be needed in order to perform deductions about the post-conditions as they may be more complicated in the proof in $\mathcal{R}$ that in that of $\mathcal{T}$ (they are used to transform time-formulas and may for that reason contain more information).

The actual formulation of the proof system $\mathcal{R}$ is to some extent a matter of taste. We can, of course, change the syntax of the formulas $P(\mathbf{c}: \mathcal{R})Q/V$ without problems but it is also possible to separate
the proof system into two parts, the one being as \( \mathcal{T} \) and the other being concerned with run-time properties alone. The formulas will thus have two different forms, for instance \( P(c > Q/V) \) and \( P(c > R/V) \) where the meaning of the latter is that if \( P \) holds then \( c \) terminates and \( R \) holds for the run-time (relative to the initial state). It is straightforward to reformulate the axioms and rules of the proof system \( \mathcal{T} \); the rule \( /; \mathcal{T}/ \) will, for instance, give rise to the following two rules:

\[
\begin{align*}
\frac{P(c_1)P'(A_{Q_1}/V, \ P'(c_2)Q_2/V)}{P(c_1;c_2)Q_1.Q_2/V} \\
\frac{P(c_1)R_1/V, \ P'(c_2)A_{Q_1}/V, \ P'(c_2)R_2/V}{P(c_1;c_2)R_1\theta(Q_1.R_2)/V}
\end{align*}
\]

Note that the first of these rules is \( /; /\mathcal{T}/ \).

The idea of splitting the proof system into parts concerned with different parts of properties can be pursued even further. The proof system \( \mathcal{T} \) for total correctness can be used to prove correctness and termination of programs, and it can be split into two parts, the one being a proof system for partial correctness and the other being concerned with termination assertions. Such a separation of concerns is not common in Hoare-like proof systems and we have therefore followed the usual practice and have extended the axioms and rules of the total correctness proof system to include deductions about the run-time properties.

In Chapter 1 we reviewed some alternative approaches to proving run-time properties of simple while programs. Two of these approaches, that of programmed counters and Shaw's method, have been considered in
Section 3.6, and we shall now briefly comment on the approaches for loop programs. Meyer and Ritchie's approach /MeRi67/ is mainly interested in the structure of the nesting of loops and is thus quite different from the ideas behind $\mathcal{R}$. Kasai and Adachi present a more refined method for the simple loop programs in /KaAd80/ but also here the structure of the nesting of loops is the important factor. The third method, that of Adachi, Kasai and Moriya /AKM80/, uses a program transformation similar to that of the method of programmed counters and then an algorithm will be used to bound the values of the so-called control variables. The application of this algorithm can be compared with a proof in the total correctness proof system with the restriction that only upper bound properties can be expressed in the pre- and post-conditions. So the drawbacks of this approach are similar to those of the method of programmed counters (see Section 3.6).

In ordinary programming languages one will often find variants of the programming constructs of the while language considered here, for example IF b THEN c, REPEAT c UNTIL b and FOR x=e_1 TO e_2 DO c. We shall conclude this chapter by suggesting an extension of the proof system $\mathcal{R}$ with rules for these constructs.

The semantics and run-time requirements of the one-armed conditional IF b THEN c can easily be specified by extending the language with another construct SKIP that does nothing and requires no time. Then IF b THEN c will be equivalent to the statement IF b THEN c ELSE SKIP. It is easy to verify that an extension of $\mathcal{R}$ with the axiom

$$P(\text{SKIP}; \text{time}=0) \vdash \text{void}$$

will be sound and complete as before. We will then have the following derived rule.
The semantics and run-time requirements of the construct
\texttt{REPEAT c UNTIL b} are equivalent to those of the program \texttt{c;WHILE \texttt{~b DO c.}}

We shall therefore suggest the following rule
\[
\begin{align*}
\text{P}(z+1) &\Rightarrow \text{P}(z) \land \text{Q}' \rightarrow \text{V}\{z\}, \quad \text{P}(0) \Rightarrow \text{b}, \quad \text{Q}' \land \text{b} \Rightarrow \text{Q}, \\
\text{Q}' &\Rightarrow \text{Q}, \quad \text{R}' \land (\text{Q}' \land \text{b}) \Rightarrow \text{R}, \quad \text{R}' \land \text{Q}' \Rightarrow (\text{E}(\text{b}) \land \text{R}) \Rightarrow \text{R} \\
\exists z. \text{P}(z+1) &\Rightarrow \text{REPEAT c UNTIL b} \Rightarrow \text{Q/V}
\end{align*}
\]
where \( z \) is a variable of sort \texttt{nat} satisfying \( z \not< \text{V} \).

As in the rule for the while construct the idea is that the value of \( z \) bounds the number of times the loop is executed. Initially, the body is executed at least once and this is expressed by the pre-condition \( \exists z. \text{P}(z+1) \) of the conclusion of the rule.

For the sake of simplicity we shall consider a variant of the \texttt{FOR-loop} \texttt{FOR x=e}_1 \texttt{ TO e}_2 \texttt{ DO c} where the value of \( x \) cannot be changed by \( c \). We can then assume that the semantics and run-time requirements of the construct are equivalent to those of the program
\[
x:=e_1; \quad y:=e_2; \quad \text{WHILE } x<y \text{ DO } (c; \quad x:=x+1) \quad (y \not< \text{FV}(c) \cup \{x\}).
\]
Based on this we shall suggest the following rule for the run-time analysis of the \texttt{FOR-loop}:
\[
\begin{align*}
x \land y \land \text{P}(x) &\Rightarrow \text{R} \Rightarrow \text{P}(x+1) \land \text{Q}' \rightarrow \text{V}\{y\}, \quad \neg(x \land y) \land \text{I}_{\text{V}}, \quad \text{Q}' \Rightarrow \text{Q} \\
\neg(x \land y) &\land E(x \land y) \Rightarrow \text{R} \quad E(x \land y) \Rightarrow (\text{E}_2(x+1) \land \text{R}) \Rightarrow \text{R} \\
P(e_1) &\Rightarrow \text{FOR x=e}_1 \text{ TO e}_2 \text{ DO } c; \quad (e_1 \land e_2 \land \neg x \land y \land \text{Q}' \rightarrow \text{V})
\end{align*}
\]
where \( x \not< \text{FV}(c) \) and \( y \) has sort \texttt{nat} and satisfies \( y \not< \text{V} \).

The idea is here that, the pre-condition \( P \) expresses that we already have considered the interval from \( e_1 \) to the current value of \( x \).
4 RUN-TIME ANALYSIS OF NON-RECURSIVE PROCEDURE PROGRAMS

The proof system $\mathcal{R}$ developed for the while language in the previous chapter is far from sufficient if one wants to formalise a larger class of the algorithm analyses found in textbooks such as /AHU74/ and /AHU82/. There are several reasons for this and the purpose of this chapter and the remaining ones is to obtain a more satisfactory proof system.

Maybe the most obvious weakness of the proof system $\mathcal{R}$ is that it only applies to while programs. Many of the interesting algorithms tend also to use interesting programming constructs such as recursive procedures. In this chapter we shall extend the while language with constructs for procedure declaration and calls. We shall allow procedures to have call-by-value and call-by-reference parameters. In the case of call-by-value a formal parameter of a procedure is treated as a local variable initialised to the value of the actual parameter. This parameter mechanism can be found in programming languages such as Pascal and Algo160. In the case of call-by-reference we shall assume that the actual parameter is a variable and we shall bind the formal parameter directly to it. This version of the call-by-reference parameter mechanism can be found in for instance Pascal and is here called call-by-variable. For the sake of simplicity we shall assume that each procedure has exactly two parameters, the one is call-by-value and the other is call-by-reference. Furthermore, we shall in this chapter restrict our attention to non-recursive procedures; a recursive version
of the language is considered in Chapter 5.

The run-time analysis of programs with non-recursive procedures is usually quite straightforward. Aho, Hopcroft and Ullman describe it as follows in /AHU82 p24/:

"If we have a program with procedures, none of which is recursive, then we can compute the running time of the various procedures one at a time, starting with those procedures that make no calls to other procedures (...). There must at least be one such procedure, else at least one procedure is recursive. We can then evaluate the running time of procedures that call only procedures that make no calls, using the already-evaluated running times of the called procedures. We continue this process, evaluating the running times of each procedure after the running times of all procedures it calls have been evaluated."

In the case of recursive procedures the analysis is more complicated and usually involves construction and solution of recurrence relations.

As mentioned we shall in this chapter restrict our attention to programs with non-recursive procedures; the more interesting case of recursive procedures is left for the next chapter. Thereby we postpone a couple of problems for a while. First, as in the previous chapter, we shall extend a proof system for total correctness to prove run-time properties. In the case of non-recursive procedures such a proof system can easily be constructed but it is well-known that if recursion is allowed it will be much more complicated. Secondly, as mentioned above, the run-time analysis can be expected to be more complicated for recursive procedure programs than for the non-recursive ones.

The syntax and semantics of the new language is presented in Section 4.1 together with its run-time requirements. The proof system $\mathcal{R}^N$ for analysis of run-time properties of programs in the language is developed in Section 4.2 and in the sections 4.3 and 4.4, respectively.
we prove soundness and completeness properties for it. Then, in Sec-
tion 4.5, we shall consider a worked example: we shall analyse an
algorithm solving the union-find problem. Finally, Section 4.6 con-
tains some concluding remarks, among others a review of three differ-
ent strategies for proving run-time properties in a proof system as 
$\mathbb{R}^N$.

4.1 THE PROCEDURE LANGUAGE

Formally, the procedure language will be parameterised on a data
type specified by a K-sorted signature $\Sigma$ (just as the while language
in Chapter 2). As in the previous chapters we shall assume that we
have given a K-sorted set $X$ of program variables and that $X$ contains
a countable infinite number of variables of each sort $k$. Furthermore,
we shall assume that we have a set $\Pi$ of procedure names. For later
use we shall assume that $\Pi$ is a countable infinite set.

The syntax of the language of procedure programs is as follows
- $x:=e$ is a procedure program
- if $c$ and $c'$ are procedure programs then so are IF $b$ THEN $c$ ELSE $c'$,
  $c;c'$, WHILE $b$ DO $c$, LET $x=e$ IN $c$ and PROC $p(VAL x,VAR y)$ IS $c$ IN $c'$
- CALL $p(e,y)$ is a procedure program.

As in the previous chapters, $x$ and $y$ are variables, $e$ is a term and
$b$ is a boolean expression. Furthermore, $p$ is a procedure name. A
typical procedure program will be denoted $c$.

The set of free variables of a procedure program $c$ is denoted $FV(c)$
and for the new constructs it is defined as follows:

$FV(LET \ x = e \ \ IN \ \ c) = FV(e) \cup (FV(c) \setminus \{x\})$
FV(PROC p(VAL x, VAR y) IS c IN c') = (FV(c) \{x, y\}) \cup FV(c')

FV(CALL p(e, y)) = FV(e) \{y\}.

For the remaining constructs the definition is as in Section 2.1. The set of free procedure names of a procedure program c is denoted FP(c). The interesting clauses of its definition are

FP(PROC p(VAL x, VAR y) IS c IN c') = FP(c) \cup (FP(c') \{p\})

FP(CALL p(e, y)) = \{p\}.

Note, it is reflected that procedures are supposed to be non-recursive.

A procedure program is well-formed if it is well-typed and if calls of procedures only happen within the scope of their declaration. In order to define this more precisely, we shall introduce a predicate WF(env, c); the intuition being that this predicate holds on env and c if the procedure program c is well-formed in the environment env.

An environment env is a finite mapping of procedure names to closures. Since the semantics is given operationally we shall define a closure to be a triple of the form (x, y, c) where x and y are the two variables being the formal parameters of the procedure and c is its body. We shall write DOM(env) for the domain of the mapping env, that is, DOM(env) \subseteq \Pi. The environment with empty domain is denoted () and we write env(p=(x, y, c)) for the environment that is as env except that the procedure name p now have associated the closure (x, y, c). The closure associated with p in env is denoted env(p).

Define the relation \(\subseteq\) on DOM(env) by

\(q \subseteq p\) if and only if env(p)=(x, y, c) and \(q \in FP(c) \{p\}\).

We shall say that env is a reasonable environment if there are no sequence \(p_1, \ldots, p_k\) of procedure names from DOM(env) satisfying \(p_i \subseteq p_{i+1}\) for \(1 \leq i < k\) and furthermore \(p_k \nsubseteq p_k\). This means that we can find
a linear ordering on the procedure names of DOM(env) satisfying that each procedure can only call procedures occurring before it in the ordering or itself.

The predicate WF(env, c) is defined by the following set of axioms and rules which should be straightforward to understand.

The predicate WF(env, c)

\[
\begin{align*}
\text{WF}(env, x:=e) & \quad \text{if } x \text{ and } e \text{ have the same sort and } env \text{ is reasonable} \\
\text{WF}(env, IF \ b \ \text{THEN } c_1 \ \text{ELSE } c_2) & \quad \text{WF}(env, c_1), \ \text{WF}(env, c_2) \\
\text{WF}(env, c_1; c_2) & \quad \text{WF}(env, c_1), \ \text{WF}(env, c_2) \\
\text{WF}(env, WHILE \ b \ \text{DO } c) & \quad \text{WF}(env, c_1), \ \text{WF}(env(p=(x,y,c_1)), c_2) \\
\text{WF}(env, LET \ x=e \ \text{IN } c) & \quad \text{WF}(env(p)=(x',y',c)) \text{ where } x' \text{ and } e \text{ have the same sort and } y' \text{ and } y \text{ have the same sort, } env \text{ is reasonable} \\
\end{align*}
\]

Note, the rule for procedure declaration reflects that procedures are non-recursive.
The semantics and run-time requirements of the procedure language will be defined operationally by extending the relation \((c,s) \xrightarrow{\mathcal{E}} s'\) introduced in Section 3.1 for the while language. We shall now consider the new constructs of the language and motivate the necessary extensions.

The construct \(\text{LET } x=e \text{ IN } c\) declares an initialised local variable \(x\) and its scope. The idea is that the value of a global variable with the same name \(x\) will not be changed by this statement. This effect can be described formally by introducing locations as is usually done in denotational semantics (see for instance /St77/). Another possibility is to rename the declared variable and in this way distinguish it from the global one. This is the approach we shall take and, as we shall see later, one of the advantages will be that the rule defining the semantics and run-time requirements for the construct will have some resemblance with the proof rule we are inventing for it. The idea of choosing the semantic description of a programming construct to be very similar to a proof rule for it also occurs in /Cl79/ and /ApdB77/.

Given a procedure program \(c\) let \(c'\) be the program obtained from \(c\) by substituting \(y\) for every free occurrence of the variable \(x\) and, at the same time, rename bound variables to avoid name clashes. We shall here require that the variables \(x\) and \(y\) have the same sort. The exact definition of the substitution is very similar to that used in lambda calculus (see for instance /HLS72/ or /St77/). We shall not give the complete definition here but merely mention a few of the interesting clauses:
\[(\text{LET } x = e \text{ IN } c) \quad \overset{Y}{\overset{X}{\equiv}} \quad \text{LET } x = e^Y \overset{X}{\text{ IN }} c,\]

\[(\text{LET } x' = e \text{ IN } c) \quad \overset{Y}{\overset{X}{\equiv}} \quad \text{LET } x' = e^Y \overset{X}{\text{ IN }} c^{Y^X} \quad \text{if } x \not= x' \text{ and either } x' \not= y \text{ or } y \not\in \text{FV}(c),\]

\[(\text{LET } y = e \text{ IN } c) \quad \overset{Y}{\overset{X}{\equiv}} \quad \text{LET } y' = e^Y \overset{X}{\text{ IN }} c^{Y'^X} \quad \text{if } x \not= y \text{ and } y \not\in \text{FV}(c); \text{ here } y' \text{ is a variable of the same sort as } y \text{ satisfying } y' \not\in \text{FV}(e) \cup \text{FV}(c).\]

In order to perform an appropriate renaming of the declared variable in the LET-construct we shall extend the relation \(\langle c, s \rangle \overset{L}{\rightarrow} s'\) of Section 3.1 to specify a finite set \(V\) of program variables whose scope might include the program \(c\), that is, we will have \(\text{FV}(c) \subseteq V\). We shall write \(V \cdot \langle c, s \rangle \overset{L}{\rightarrow} s'\) for the new relation. The semantics and run-time requirements of the LET-construct can now be specified by the following rule:

\[
\frac{V \cdot \langle c, s \rangle \overset{L}{\rightarrow} s'}{V \cdot \langle \text{LET } x = e \overset{L}{\text{ IN }} c, s \rangle \overset{L}{\rightarrow} s'}
\]

where \(x'\) is a variable of the same sort as \(x\) satisfying \(x' \not\in V\).

Note that we use the set \(V\) when choosing a new variable for the renaming of the declared variable. The renaming will always be possible since we have assumed that we have a countable infinite set of variables of each sort. In the computational model we have neglected the time that might be required to allocate space for the new variable but the rule can easily be modified to reflect more refined definitions of computational models.

The construct \(\text{PROC } p(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 \text{ IN } c_2\) declares a local procedure \(p\) with body \(c_1\) and scope \(c_2\). The problem of possible name clashes can, as in the case of variable declarations, be solved by renaming the declared procedure. As before, one of the benefits of
solving the problem in this way is that the rule defining the semantics and run-time requirements of the construct resembles the proof rule to be developed later.

For a procedure program $c$ we define $c^q_p$ to be the program that is as $c$ except that all free occurrences of calls of the procedure $p$ are replaced by calls of $q$ and the bound procedure names are renamed if necessary. The formal definition of $c^q_p$ is very similar to that of $c^x_p$ above with the interesting clauses being

\[
\text{(PROC } p(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 \text{ IN } c_2^q_p \text{)}
\]

\[
\text{PROC } p(\text{VAL } x, \text{VAR } y) \text{ IS } c^q_1 \text{ IN } c_2^q_p
\]

\[
\text{PROC } p'(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 \text{ IN } c_2^q_p
\]

\[
\text{PROC } p'(\text{VAL } x, \text{VAR } y) \text{ IS } c^q_1 \text{ IN } c_2^q_{1p}
\]

\[
\text{if } p \not\equiv p' \text{ and either } p \not\equiv q \text{ or } q \not\equiv \text{FP}(c_2),
\]

\[
\text{(PROC } q(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 \text{ IN } c_2^q_p \text{)}
\]

\[
\text{PROC } q'(\text{VAL } x, \text{VAR } y) \text{ IS } c_1^q \text{ IN } c_2^q_{1p}
\]

\[
\text{if } p \not\equiv q \text{ and } q \not\equiv \text{FP}(c_2); \text{ here } q' \text{ is a procedure name satisfying } q' \not\equiv \text{FP}(c_1) \text{ if } q \not\equiv \text{FP}(c_2),
\]

\[
\text{(CALL } p(e,y)^q_p \equiv \text{CALL } q(e,y),
\]

\[
\text{(CALL } p'(e,y)^q_p \equiv \text{CALL } p'(e,y) \text{ if } p \not\equiv p'.
\]

In order to define the semantics and run-time requirements of the procedure declaration and the procedure call-statements we shall extend the relation $\langle v,c,s \rangle \Rightarrow_s s'$ to specify the current environment $\text{env}$. The new relation will be written $(V,\text{env}) \langle v,c,s \rangle \Rightarrow_{s'} s'$. The role of the component $\text{env}$ is twofold. First, it is used to keep a record of the procedures declared so far, this is important for the handling of procedure calls. Furthermore, its domain $\text{DOM}(\text{env})$ is used in connection with the renaming of declared procedures, exactly as the set $V$
is used for renaming declared variables.

For the procedure declaration statement we have the following rule defining its semantics and run-time requirements:

\[(V,\text{env}(q=(x,y,c_1)))\vdash c \overset{q}{\rightarrow} s \Rightarrow s',\]

\[(V,\text{env})\vdash \langle \text{PROC } p(\text{VAL } x,\text{VAR } y) \text{ IS } c_1 \text{ IN } c_2, s \overset{\cdot}{\rightarrow} s' \]

where q is a procedure name satisfying \(q \in \text{DOM(env)}\).

This rule reflects that procedures are non-recursive: a call of p in \(c_1\) will refer to a previously declared procedure named p and thus the call of p should not be replaced by a call of q. Since we have a countable infinite set of procedure names the renaming will always be possible. In the definition of a computational model we have not accounted for the time it may take to record the declaration of a new procedure but the rule above can easily be modified to reflect more refined models.

In the environment env the statement CALL p(e,y) is, intuitively, equivalent to the composite statement LET \(x' = e \text{ IN } c_y\), where env(p)= (\(x',y',c\)) - remember that the first parameter is call-by-value and the second is call-by-variable. Using this observation we shall define the semantics and run-time requirements of the procedure call by the following rule

\[(\forall \{x\}, \text{env})\vdash \langle c_{x,x'}, s_{x}^e(s) \rangle \overset{\cdot}{\rightarrow} s'\]

\[(V,\text{env})\vdash \langle \text{CALL } p(e,y), s \rangle \overset{s_e(s) + e(s) + r}{\rightarrow} s',\]

where \(\text{env}(p) = (x',y',c)\) and x is a variable of the same sort as \(x'\) satisfying \(x' \notin \{x,y\}\).

The definition of the computational model ignores the time that might be required for, for instance, transfer of control in a procedure call.
but again, the rule can easily be modified to reflect other choices.

This completes the description of the semantics and run-time requirements of the new constructs. The complete set \( \mathcal{N} \) of axioms and rules defining the semantics and run-time requirements of the non-recursive procedure language is listed below:

Semantics and run-time of non-recursive procedure programs: \( \mathcal{N} \)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ass-( \mathcal{N} /</td>
<td>(V,env) \models \langle \text{x:=e}, s \rangle \xrightarrow{e} (s) + e(s) \xrightarrow{e} s</td>
</tr>
<tr>
<td>/IF-( \mathcal{N} /</td>
<td>\begin{align*} (V,env) \models \langle \text{IF b THEN c_1 ELSE c_2}, s \rangle &amp; \xrightarrow{b} (s) + r \xrightarrow{s} s' \ (V,env) \models \langle \text{IF b THEN c_1 ELSE c_2}, s \rangle &amp; \xrightarrow{b} (s) + r \xrightarrow{s} s' \end{align*}</td>
</tr>
<tr>
<td>/;-( \mathcal{N} /</td>
<td>\begin{align*} (V,env) \models \langle c_1, s \rangle &amp; \xrightarrow{c_1}, s \xrightarrow{c_1}, s' \ (V,env) \models \langle c_2, s' \rangle &amp; \xrightarrow{c_2}, s'' \ (V,env) \models \langle c_1; c_2, s \rangle &amp; \xrightarrow{c_1; c_2}, s \xrightarrow{c_1; c_2}, s' \xrightarrow{c_1; c_2}, s'' \end{align*}</td>
</tr>
<tr>
<td>/WHILE-( \mathcal{N} /</td>
<td>\begin{align*} (V,env) \models \langle \text{WHILE b DO c}, s \rangle &amp; \xrightarrow{b} (s) + r \xrightarrow{c}, s'' \ (V,env) \models \langle \text{WHILE b DO c}, s \rangle &amp; \xrightarrow{b} (s) + r \xrightarrow{c}, s'' \end{align*}</td>
</tr>
<tr>
<td>/LET-( \mathcal{N} /</td>
<td>\begin{align*} (V,env) \models \langle \text{LET \text{x:=e} IN c}, s \rangle &amp; \xrightarrow{\text{x:=e}}, s \xrightarrow{\text{x:=e}}, s' \ (V,env) \models \langle \text{LET \text{x:=e} IN c}, s \rangle &amp; \xrightarrow{\text{x:=e}}, s \xrightarrow{\text{x:=e}}, s' \end{align*}</td>
</tr>
</tbody>
</table>

where \( x' \) is a variable of the same sort as \( x \) satisfying \( x' \notin V \)

\( \text{(cont.)} \)
Semantics and run-time of non-recursive procedure programs: \( \mathcal{F}^N \) (cont.)

\[
/\text{PROC-}^{N}/ \quad (V,\text{env}(q=(x,y,c))) \vdash (c_{2p},s) \rightarrow s'
\]

\[
(V,\text{env}) \vdash \langle \text{PROC } p(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 \text{ IN } c_2, s \rangle \rightarrow s'
\]

where \( q \) is a procedure name satisfying \( q \in \text{DOM}(\text{env}) \)

\[
/\text{CALL-}^{N}/ \quad (V\{x\},\text{env}) \vdash (c_{x^y},s) \rightarrow s'
\]

\[
(V,\text{env}) \vdash \langle \text{CALL } p(e,y), s \rangle \rightarrow (s'+e(s)+r)'s'
\]

where \( \text{env}(p)=(x',y',c) \) and \( x \) is a variable of the same sort as \( x' \) satisfying \( x \in V \{ y' \} \).

**Properties of (Non-Recursive) Procedure Programs**

As in the previous chapters we shall list some lemmas about the semantics and run-time requirements of the language. The first two lemmas are extensions of the lemmas 3.1-1 and 3.1-2 in Chapter 3:

**Lemma 4.1-1.** If \( (V,\text{env}) \vdash (c,s) \rightarrow s' \), \( V' \cap (\text{FV}(c) \cup \text{FV}(\text{env})) = \emptyset \) and \( V' \subseteq V \)

then \( s \in \text{V}, s' \).

**Lemma 4.1-2.** If \( (V\cup V',\text{env}) \vdash (c,s) \rightarrow s' \), \( V' \cap (\text{FV}(c) \cup \text{FV}(\text{env})) = \emptyset \) and \( s \in V, s_0 \)

then \( (V,\text{env}) \vdash (c,s_0) \rightarrow s_0' \) where \( s_0 \in V, s' \) and the two proofs in \( \mathcal{F}^N \)

have the same lengths.

Here, \( \text{FV}(\text{env}) = \cup \{ \text{FV}(c) - \{ x,y \} | p \in \text{DOM}(\text{env}) \} \) and \( \text{env}(p)=(x,y,c) \). Both lemmas can straightforwardly be proved by induction on the length of the proofs in \( \mathcal{F}^N \). We omit the details.

In the soundness proof in Section 4.3 we shall use a lemma that allows us to extend the \( \text{V} \)-part of a formula \( (V,\text{env}) \vdash (c,s) \rightarrow s' \):

**Lemma 4.1-3.** If \( (V,\text{env}) \vdash (c,s) \rightarrow s' \) and \( V' \cap V = \emptyset \) then for some \( s'' \)

\( (V\cup V',\text{env}) \vdash (c,s) \rightarrow s'' \) where \( s'' \in V, s' \) and \( s'' \in V, s' \).

152
In order to prove this result we need a result about substitution of variables. For an environment env and two variables x and y of the same sort and satisfying \( y \not\in \text{FV}(\text{env}) \), we define \( \text{env}^y_x \) as follows:

\[
\text{env}^y_x \triangleq (\cdot),
\]

\[
\text{env}(p=(x',y',c))^y_x \triangleq \text{env}^y_x(p=(x',y',c)) \quad \text{if} \quad x \notin \{x',y'\},
\]

\[
\text{env}(p=(x',y',c))^y_x \triangleq \text{env}^y_x(p=(x',y',c_y)) \quad \text{if} \quad x \notin \{x',y'\},
\]

\[
\text{env}(p=(x',y',c))^y_x \triangleq \text{env}^y_x(p=(x'',y'',c_x,y',y')) \quad \text{if} \quad x \notin \{x',y'\}, y \notin \{x',y'\}
\]

and \( x'' \) and \( y'' \) are distinct variables of the same sorts as \( x' \) and \( y' \), respectively, and \( x'' \notin \text{FV}(c) \cup \text{FV}(\text{env}) \).

Then we have

**Lemma 4.1-4.** Assume that \( (V,\text{env}) \vdash \langle c, s^y_x \rangle \overset{r}{\rightarrow} s' \) and that \( x \) and \( y \) have the same sorts and satisfy \( x \in V \) but \( y \notin V \). Then for some \( s'' \)

\[
(V \cup \{y\}, \text{env}^y_x) \vdash \langle c^y_x, s \rangle \overset{r}{\rightarrow} s''
\]

where \( s'' \in (V \cup \{y\})^s_x \) and the two proofs in \( \mathcal{F}_N \) have the same lengths.

Thus the lemma allows us to rename a variable in a program \( c \). Intuitively, the substitution of \( y \) for \( x \) in the environment is needed because we have implemented static scope by dynamic scope in the semantics. The proofs of the two lemmas 4.1-3 and 4.1-4 can be found in Appendix B.

In the completeness proof of Section 4.4 we shall use the following lemma expressing that the language is deterministic:

**Lemma 4.1-5.** If \( (V,\text{env}) \vdash \langle c, s \rangle \overset{r}{\rightarrow} s' \) and \( (V,\text{env}) \vdash \langle c, s' \rangle \overset{r'}{\rightarrow} s'' \) then \( s' \equiv_s s'' \) and \( r = r' \).

The proof of this result is by induction on the length of the two proofs.
In order to prove the result in the case of a procedure declaration we shall need a lemma allowing us to rename procedures. For an environment env and two procedure names p and q with q ∈ DOM(env) we define env_q^p as follows

\[
\begin{align*}
&\text{env}^q_p(\text{p}=(x,y,c)) = \text{env}_p^q(q=(x,y,c^q)), \\
&\text{env}^q_p(p'=(x,y,c)) = \text{env}_p^q(p'=\text{p}^q) \quad \text{if p} \not\equiv p'.
\end{align*}
\]

Then we have the following result

**Lemma 4.1-6.** Assume that (V,env) ⊢ c, s → s', p ∈ DOM(env) and q ∈ DOM(env). Then (V,env_q^p_p) ⊢ c, s → s' and the two proofs in \( F^N \) have the same lengths.

The proof of this lemma as well as that of Lemma 4.1-5 can be found in Appendix B.

### 4.2 The Proof System \( R^N \)

Having defined the semantics and run-time requirements of the procedure language we now turn to the development of a proof system called \( R^N \) for analysis of run-time properties of programs in the language. The formulas of the proof system have the form P(c : R) Q / (V, env) where c is a procedure program and as in the previous proof system \( R \), P, Q and R are a pure formula, a relational formula and a time formula, respectively. The component V is a finite set of program variables as before whereas env is an environment. The role of the component V is twofold. First, it is used in the specification of the post-condition Q exactly as in the proof systems \( T \) and \( R \). Secondly, we shall use V in connection with a renaming of the declared variables that is very
similar to the one performed in the semantic rules of Section 4.1.

The environment env is intended to be the current one. It keeps a record of the declared procedures and is used in connection with procedure calls to get hold of the appropriate procedure bodies, just as in the semantic rules of Section 4.1. Furthermore, its domain is used when renaming declared procedures, again this is very much as in the semantic rules.

We shall impose a well-formedness condition on the formulas. A formula $P[c:R](V,env)$ is a well-formed formula of $\mathcal{X}$ if

- $\text{FV}(c) \subseteq V$ and $\text{FP}(c) \subseteq \text{DOM}(env)$,
- $\text{FV}(P) \subseteq V$, $\text{FV}(Q) \subseteq \text{V} \cup \{\text{time}\}$,
- $\text{FV}(env) \subseteq V$ and $\text{FP}(env) \subseteq \text{DOM}(env)$.

Here we use $\text{FV}(env)$ as an abbreviation for

$$\bigcup_{p \in \text{DOM}(env)} \{\text{FV}(c) \cap \{x,y\} \mid \text{env}(p) = (x,y,c)\}$$

and $\text{FP}(env)$ stands for

$$\bigcup_{p \in \text{DOM}(env)} \{\text{FP}(c) \mid \text{env}(p) = (x,y,c)\}.$$ 

Intuitively, the well-formedness condition ensures that the program variables that might be referenced when executing $c$ in the environment $env$ are included in the set $V$ - there might of course be locally declared variables and procedures beside those of $V$ and $env$, respectively.

Given a computational model for the data type we define the validity of a well-formed formula $P[c:R](V,env)$, written $\mathcal{V}P[c:R](V,env)$, as follows

for every state $s$ satisfying $\mathcal{V}P(s)$ there is a state $s'$ and a natural number $r$ such that

$$(V,env) \vdash c,s \xrightarrow[]{} s', \mathcal{V}Q(s,s') \text{ and } \mathcal{V}R(s,r).$$
The axioms and rules of the proof system $\mathcal{R}$ can easily be extended to record the current environment. They are listed below together with the axioms and rules for the new constructs of the language - the latter are explained afterwards.

The proof system $\mathcal{R}^N$:

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ass-}^{\mathcal{R}^N}$</td>
<td>$P(x:=e : E)(e) \vdash_{V} x = e / (V, env)$</td>
</tr>
<tr>
<td>$\text{IF-}^{\mathcal{R}^N}$</td>
<td>$P(b \Rightarrow c_1 \ 	ext{IF } b \text{ THEN } c_1 \ 	ext{ELSE } c_2 : E(b) \Rightarrow Q / (V, env)$</td>
</tr>
<tr>
<td>$\text{-}^{\mathcal{R}^N}$</td>
<td>$P(c_1 : R) \Rightarrow P'Q_1 / (V, env), \ P(c_2 : R) \Rightarrow Q_2 / (V, env)$</td>
</tr>
<tr>
<td>$\text{WHILE-}^{\mathcal{R}^N}$</td>
<td>$P(z) \Rightarrow P(z+1) \Rightarrow Q / (V, env), \ P(0) \Rightarrow b,\ P(z) \Rightarrow Q / (V, env)$</td>
</tr>
<tr>
<td>$\text{LET-}^{\mathcal{R}^N}$</td>
<td>$P(y = e : x) \Rightarrow y / (V, env)$</td>
</tr>
<tr>
<td>$\text{PROC-}^{\mathcal{R}^N}$</td>
<td>$P(\text{PROC } p(\text{VAL } x, \text{VAR } y) \ IS \ c_1 \ IN \ c_2 : R \Rightarrow Q / (V, env)$</td>
</tr>
</tbody>
</table>

(cont.)
The proof system $R^N$ (cont.)

\[
\text{CALL-$R^N$/} \quad \frac{P_{\text{x}}=e(c, y', RQ/(V, env))}{P_{\text{CALL}}(e(y), RQ/(V, env))}
\]

where $env(p)=(x', y', c)$ and $x$ is a variable of the same sort as $x'$ satisfying $x \notin \text{FV}(Q) \cup y' \cup y$. 

\[
\text{cons-$R^N$/} \quad \frac{P \Rightarrow P', P' \langle c: RQ'/V, env \rangle, Q' \Rightarrow Q, R' \Rightarrow R}{P \langle c: RQ'/V, env \rangle}
\]

\[
\text{inv-$R^N$/} \quad \frac{P \langle c: RQ'/V, env \rangle}{P \langle c: PARPQ'/V, env \rangle}
\]

We shall write $R^N \vdash P \langle c: RQ'/V, env \rangle$ if the formula $P \langle c: RQ'/V, env \rangle$ is provable using the axioms and rules above (using only the true formulas from the assertion language).

The rule /LET-$R^N$/ for variable declaration resembles the semantic rule /LET-$R^N$/ of Section 4.1. The local variable $x$ is renamed and this makes it possible to distinguish between occurrences of $x$ in the body $c$ of the construct and possible free occurrences of $x$ in the formulas $P, Q$ and $R$ - these occurrences will refer to the non-local variable $x$. Note that the condition $y \notin \text{FV}(Q)$ is not needed if the post-condition of the rule is replaced by the formula $\exists y. c_y$. The rule /PROC-$R^N$/ is the obvious one, it merely records the new environment. The rule /CALL-$R^N$/ reflects that the statement CALL $p(e, y)$ is equivalent with the program LET $x'=e$ IN $c'$, where $env(p)=(x', y', c)$. In fact, using the rule /LET-$R^N$/ to verify that from a proof of the formula

\[
P_{\text{x}}=e(c, y', RQ/(V, env))
\]

in $R^N$ we can obtain a proof of the formula
(assuming that $x \notin \text{VwFV}(Q) \cup \{y^2\}$).

**EXAMPLE**

As an example of the use of the proof system $\mathcal{R}^N$ let us consider the following program computing the $n$ first Fibonacci numbers. The program uses the data type of one-dimensional arrays (Example 3.1-3).

PROC next(VAL x,VAR a)

IS IF $x = 0$ THEN $a := \text{upd}(a, x, 1)$

ELSE $a := \text{upd}(a, x, a[x-1] + a[x-2])$

IN LET $x = 0$

IN WHILE $x \lt n$ DO (CALL next(x, fib); $x := x+1$).

Here $x$ and $n$ are variables of sort nat and fib and $a$ are variables of sort array. Using the computational model of Example 3.1-3 we shall below sketch a proof of the formula

(1) $\text{length}(\text{fib}) = n \langle \text{Fibonacci: time} \langle 27n + 5 \rangle \text{TRUE} / (\{\text{fib}, n\},())$.

First we shall use the rules of $\mathcal{R}^N$ to prove that it is sufficient to construct a proof for the while loop of the formula

(2) $\text{length}(\text{fib}) = n \langle x = 0 \langle \text{WHILE } x \lt n \text{ DO ( . . . ) : time} \langle 27n + 3 \rangle \text{TRUE} / (\{\text{fib}, n, x\}, \text{env})$

where env is the environment (next = ($x, a$, IF $x = 0$ THEN $...$ ELSE $...$)). To see this we shall first apply the rule /LET-$\mathcal{R}^N$/ to (2). Without loss of generality, we can assume that $E^S(0)$ is the formula $\text{time}=2$ so we get a proof of

$\text{length}(\text{fib}) = n \langle \text{LET } x = 0 \text{ IN WHILE } ... \text{ DO } ... : (\text{time}=2) \Theta (\text{time} \langle 27n + 3 \rangle) \rangle$

$\text{TRUE}_x^0 / (\{\text{fib}, n\}, \text{env})$.  

158
Using /PROC-\mathcal{X}^{N}/ we then get a proof of

\[
\text{length}(fib) = n \langle \text{Fibonacci; } (time=2) \to (time<27' n+3) \to \text{TRUE}/\{\{fib, n\}, (\text{env})\}. 
\]

Since \( \textbf{F}(time=2) \to (time<27' n+3) \to time<27' n+5 \) we have the required proof of (1).

Let us now construct the proof for the while loop. The invariant \( P(z) \) is chosen to be the formula \( n = z + x \land \text{length}(fib) = n \). For a moment assume that we have a proof of the following property of the procedure call:

(3) \( P(z+1) \land x < n \langle \text{CALL next}(x, fib); time<20 \rangle P(z+1) \land \overline{x} \land \overline{an} = \overline{n} / (\{\text{env}\}) \)

where \( V = \{\text{fib, n, x, z}\} \). It is then straightforward to construct a proof of the formula

\[
P(z+1) \land x < n \langle \text{CALL next}(x, fib); x := x+1; \text{time<24} \rangle P(z) \land \overline{x} \land \overline{an} = \overline{n} / \\
(V, \text{env})
\]

(using that \( \textbf{E}^{S}(x+1) \) can be chosen to be \( \text{time}=4 \)). It is easy to verify that \( \textbf{F}(P(0) \to \neg(x<n) \land P(z) \land \neg(x<n) \land \{\text{fib, n, x}\} \to \text{TRUE} \) and

\( \textbf{F}(x = x+1 \land \overline{an} = \overline{n}) \to \text{TRUE} \to \text{TRUE} \). The time invariant \( R \) for the loop is chosen to be the formula \( \text{time}+27' x < 27' n+3 \). Since \( E(x<n) \) can be chosen to be the formula \( \text{time}=3 \) we have \( \textbf{F}(z) \land \neg(x<n) \land E(x<n) \to R \) and

\( \textbf{F}(x<n) \lor (\text{time<24}) \lor ((x = x+1 \land \overline{an} = \overline{n}) \to R) \). So the rule /WHILE-\mathcal{X}^{N}/ can be applied and we get a proof of

\[
\exists z. P(z) \langle \text{WHILE } x<n \text{ DO } ... : R \rangle \text{TRUE}/\{\{fib, n, x\}, \text{env}\}.
\]

We have \( \textbf{F}\text{length}(fib) = n \to \exists z. P(z) \) and \( \textbf{F}\text{length}(fib) = n \land x = 0 \land \text{time<27' n+3} \)
so using the general rules we get the required proof of (2).

We now turn to a proof for the procedure call, more precisely, a proof of (3). It is here sufficient to prove the formula
To see this we first observe that $E^S(x)$ can be chosen to be $\text{time} = 2$ and that $(\text{time} = 2) \theta (\text{time} \leq 18) \rightarrow \text{time} \leq 20$. Furthermore, $P(z+1)Ax < nAx = \bar{n}Az = \bar{z} \rightarrow P(z+1)Ax = \bar{n}Ax = \bar{n}$. The formal parameter $x$ of the procedure has been renamed to $x'$ since we already have a variable named $x$. With these things in mind it is straightforward to verify that (3) follows from (4) using the rule $/\text{CALL}-\mathcal{R}^N/$ and the general rules. The proof of (4) is straightforward and is therefore omitted.

4.3 THE SOUN N ESS THEOREM FOR $\mathcal{R}^N$

In this section and the next one we shall consider the theoretical properties of the proof system $\mathcal{R}^N$. The soundness result which ensures that anything that is provable in $\mathcal{R}^N$ does indeed hold is as follows:

The Soundness Theorem for $\mathcal{R}^N$

Given a data type and a numerical computational model for it, if the time expressiveness condition is fulfilled then for every well-formed formula $P(c:RQ/ (V, \text{env})$ of $\mathcal{R}^N$

$$\mathcal{R}^N \vdash P(c:RQ/ (V, \text{env})) \implies \forall P(c:RQ/ (V, \text{env})).$$

In order to prove this result it is sufficient to prove that the axiom of $\mathcal{R}^N$ is valid and that the rules of $\mathcal{R}^N$ preserve validity. It is straightforward to extend the proof in Section 3.3 showing that the axiom $/\text{ass}-\mathcal{R}/$ is valid to prove that $/\text{ass}-\mathcal{R}^N/$ is valid so we omit the details here. The proofs showing that the rules $/\text{IF}-\mathcal{R}/$, $/\text{inv}-\mathcal{R}/$, $/\text{cons}-\mathcal{R}/$ and $/\text{inv}-\mathcal{R}/$ preserve validity are straightforward extensions of those in Section 3.3 for the corresponding rules of $\mathcal{R}$ so also here.
we shall omit the details. A complication arises in the proof for the rule /WHILE-\(R^N\) compared with that for /WHILE-\(\mathcal{R}\) in Section 3.3 because the component \(V\) of the formulas now has two roles so we shall sketch the proof for this rule below. Furthermore, we shall give the proofs for the new rules /LET-\(R^N\)/, /PROC-\(R^N\)/ and /CALL-\(R^N\)/.

Case /WHILE-\(R^N\)/: We shall prove that the rule preserves validity so assume that

\[ \forall P(z+1) \land c : R' \rightarrow P(z) \land Q'/(\forall u(z), \env) \]

and furthermore that \( \exists P(0) \rightarrow \neg b, \forall P(z) \land b \land I \rightarrow Q, \forall Q' \rightarrow Q \), \( \forall P(z) \land b \land E(b) \rightarrow R \) and \( \forall E(b) \rightarrow R(Q'^{-}R) \rightarrow R \), where \( z \) is a variable of sort \( \text{nat} \) satisfying \( z \notin V \). We shall prove that

\[ \forall z. P(z) \langle \text{WHILE } b \text{ DO } c : R \rangle Q/(V, \env) \]

so assume that \( \exists z. P(z)(s) \) holds for some state \( s \), that is, \( P(z)(s^n) \) for some natural number \( n \). By induction on \( n \) we shall prove that for some state \( s' \) and natural number \( r \)

\[ (V, \env) \langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow s', \exists Q(s, s') \) and \( \forall R(s, r) \).

The case where \( n=0 \) is a straightforward modification of the similar case in the proof showing that /WHILE-\(\mathcal{R}\)/ preserves validity (see Section 3.3). We omit the details here. For the induction step assume that

\( (\$) \) holds for \( n=n' \) and we shall prove it for \( n=n'+1 \). The case where \( \exists b(s) \) holds is similar to the case where \( n=0 \) and is therefore omitted. So assume that we in addition to \( P(z)(s^{n'+1}) \) have \( \exists b(s) \). Since \( z \notin FV(b) \) we have \( P(z+1) \land b(s^{n'}) \) and from (1) we get that for some \( s' \) and \( r' \)

\[ (\forall u(z), \env) \langle c, s \rangle \rightarrow r' s', P(z) \land Q'(s^{n'} \land s'), \exists Q'(s^{n'} \land s'), \forall R'(s^{n'} \land s'), \forall R(s, r) \].

From \( z \notin FV(c) \cup FV(\env) \) and Lemma 4.1-1 we get \( z(s')=n' \) so we have

\[ P(z)(s'^{n'}) \]. The induction hypothesis then gives
for some \( s'' \) and \( r'' \). Since \( z \notin \text{FV}(c) \cup \text{FV}(\text{env}) \) we can apply Lemma 4.1-2 to (2) and get

\[
(V, \text{env}) \vdash \langle \text{WHILE } b \text{ DO } c, s' \rangle \xrightarrow{r} s'' \quad \text{where } s'_0 \equiv s'
\]
since \( s' \equiv s''^n \). Lemma 4.1-2 applied to (3) gives

\[
(V, \text{env}) \vdash \langle \text{WHILE } b \text{ DO } c, s'_0 \rangle \xrightarrow{r} s'' \quad \text{where } s'_0 \equiv s''
\]
since \( s'_0 \equiv s'' \). So \( /\text{WHILE}^-/ \) gives

\[
(V, \text{env}) \vdash \langle \text{WHILE } b \text{ DO } c, s \rangle \xrightarrow{b(s)+r'+r''} s''_0.
\]
The proofs showing that \( \text{Q}(s, s'') \) and \( \text{R}(s, b(s)+r'+r'') \) hold are essentially as in the proof showing that \( /\text{WHILE}^-/ \) preserves validity (see Section 3.3) so we omit the details.

Case \( /\text{LET}^-Y^-/ \): We shall prove that the rule preserves validity so assume that

\[
(1) \quad \text{FP}(x=e) \vdash c_y : e^y \text{R}^{(V,v)} ((V \cup \{y\}, \text{env})
\]
where \( y \) is a variable of the same sort as \( x \) satisfying \( y \notin \text{V} \cup \text{FV}(Q) \). To prove

\[
\text{FP} \langle \text{LET } x=e \text{ IN } c : e^y \rangle (V, \text{env})
\]
assume that \( \text{FP}(s) \) holds for some state \( s \). Since \( y \notin V \) and \( FV(P) \subseteq V \) we have \( \text{FP}(x=e^y \text{e}(s)) \) so from (1) we get that for some state \( s' \) and natural number \( r \)

\[
(V \cup \{y\}, \text{env}) \vdash \langle c_y \circ e^y \rangle \xrightarrow{r} s', \text{Q}(s_y^e(s), s'), \text{R}(s^e(s), r).
\]
The semantic rule \( /\text{LET}^-Y^-/ \) now gives

\[
(V, \text{env}) \vdash \langle \text{LET } x=e \text{ IN } c, s \rangle \xrightarrow{e^y(s)+e(s)+r} s'.
\]
From \( \text{Q}(s_y^e(s), s') \) we get \( \text{Q}_y^e(s, s') \). The time expressiveness condition together with Lemma 3.2-1 gives that \( \text{TE}(s, e^y(s)+e(s)+r) \) holds.
From $\text{P}_R(s,r)$ we get $\text{P}_R^e(s,r)$ and thereby $\text{P}_E^e(s) \& \text{P}_R^e(s) + e(s)^+ + r$ as required. This proves the result.  

Case /PROC/: We shall prove that the rule preserves validity so assume that

(1) $\text{P}(c_{2p}:R/o/(V,env)(q=(x,y,c_1)))$

holds for $q \in \text{DOM}(env)$. To prove

$\text{P}(\text{PROC} p(\text{VAL} x,\text{VAR} y) \text{ IS } c_1 \text{ IN } c_2:R/o/(V,env))$

assume that $\text{P}(s)$ holds for some state $s$. From (1) we then get

$\langle c_{2p},s \rangle \Rightarrow s', Q(s,s')$ and $\text{P}_R(s,r)$

for some state $s'$ and natural number $r$. The semantic rule /PROC-\(^N\) now gives

$\langle c_{2p},s \rangle \Rightarrow \text{PROC} p(\text{VAL} x,\text{VAR} y) \text{ IS } c_1 \text{ IN } c_2:R/o/(V,env)$

and since $Q(s,s')$ and $\text{P}_R(s,r)$ hold the result follows immediately.

Case /CALL-\(^N\)/: We shall prove that the rule preserves validity so assume that

(1) $\text{P}(\text{CALL} p(e,y):E^s(e) \& R_{x}^e(s) + e(s)^+ + r)$

where $\text{env}(p)=(x',y',c)$ and $x$ is a variable of the same sort as $x'$ satisfying $x \notin FV(Q \cup \{y\})$. To prove

$\text{P}(\text{CALL} p(e,y):E^s(e) \& R_{x}^e(s) + e(s)^+ + r)$

assume that $\text{P}(s)$ holds for some state $s$. Then $\text{P}(s) = e(s_x)$ holds since $FV(P) \subseteq V$ and $x \notin V$ so from (1) we get that for some state $s'$ and natural number $r$

$\langle c_{x},y, s_x^e(s),r \rangle \Rightarrow s', Q(s_x^e(s),s')$ and $\text{P}_R(s_x^e(s),r)$

Now the semantic rule /CALL-\(^N\) gives:
From $\exists Q(s, s')$ we get $\exists Q(s, s')$. The time expressiveness assumption together with Lemma 3.2-1 gives $\exists E(e)(s, e(s)+e(s)+r)$. From $\exists R(s, r)$ we get $\exists R(s, r)$ and thereby $\exists E(e) \exists R(s, r) + e(s)+r$ as required.

This proves the soundness result for $\mathcal{R}^N$.

4.4 THE COMPLETENESS THEOREM FOR $\mathcal{R}^N$

The completeness result for the proof system $\mathcal{R}$ in Chapter 3 is obtained under the assumption that the input/output relation as well as the input/run-time relation for every while program is expressible by a formula of the assertion language. This assumption will now be modified such that it holds for every procedure program and we shall in this section show that if this expressiveness condition is fulfilled then we obtain a completeness result for the proof system $\mathcal{R}^N$.

Given a data type and a numerical computational model for it we say that the expressiveness condition for $\mathcal{R}^N$ is fulfilled if

- the time expressiveness condition is fulfilled,
- for every procedure program $c$, every finite set $V$ of program variables and every reasonable environment $env$ with $FV(c) \cup FV(env) \subseteq V$ and $FP(c) \cup FP(env) \subseteq \text{DOM}(env)$ there exists a relational formula $G_{V, env}[c]$ with $FV(G_{V, env}[c]) \subseteq V$ and satisfying that for every pair $(s, s')$ of states

$$ \exists G_{V, env}[c](s, s') $$

if and only if

$$(V, env) \vdash \langle c, s \rangle \xrightarrow{e} s'' \text{ for some state } s'' \text{ with } s'' \in V, s'$$

164
and there exists a time formula \( E_{V,\text{env}}[c] \) with \( \text{FV}(E_{V,\text{env}}[c]) \subseteq V \{\text{time}\} \) and satisfying that for every pair \((s, r)\) of state and natural number

\[
E_{V,\text{env}}[c](s, r)
\]

if and only if

\[(V, \text{env}) \vdash \langle c, s \rangle^T \rightarrow s' \text{ for some state } s'.\]

In the expressiveness condition for \( R \) defined in Section 3.4 we require that the formulas \( G[c] \) and \( E[c] \) exist for every while program \( c \). Since the effect of executing a program in the procedure language depends on the environment it is necessary to let the formulas depend on this as well. It is not really necessary to let the formulas depend on the set \( V \) of variables. In the definitions of \( G_{V,\text{env}}[c] \) and \( E_{V,\text{env}}[c] \) above we can, without problems, replace \( V \) by the set \( \text{FV}(c) \cup \text{FV}(\text{env}) \). The two sets of definitions will be equivalent and since the one given above is the most convenient one in the proofs below we shall prefer it.

**The completeness result and its proof**

Using this notion of expressiveness we have the following result

**The Completeness Theorem for \( R^N \)**

Given a data type and a numerical computational model for it, if the expressiveness condition for \( R^N \) is fulfilled then for every well-formed formula \( P \langle c: R^Q \rangle/(V, \text{env}) \) of \( R^N \):

\[
P \langle c: R^Q \rangle/(V, \text{env}) \text{ implies } R^N \vdash P \langle c: R^Q \rangle/(V, \text{env}).
\]

The proof of this result is slightly complicated. A simple structural induction on the program \( c \) similar to that used in the
completeness proof for $R$ in Chapter 3 will not work here because of the possibility of several procedures calling each other. We shall therefore extend the structural induction with an induction on the depth of these calls. Since the procedures are not allowed to call each other recursively we can bound the maximal depth of procedure calls statically.

Given a reasonable environment $env$ we shall define a mapping $\tilde{env}$ that to each procedure name $p$ of $\text{DOM}(env)$ associates a natural number denoted $\tilde{env}(p)$ bounding the depth of procedure calls that might be invoked when the body of the procedure $p$ is executed in the environment $env$. Without loss of generality we can assume that $env$ has the form $(p_1=(x_1,y_1,c_1)) \cdots (p_m=(x_m,y_m,c_m))$ where $p_i \not\leq p_j$ implies $i < j$ (remember, the ordering $\not\leq$ on $\text{DOM}(env)$ exists because $env$ is reasonable). We shall now define $\tilde{env}$ by

$$
\tilde{env}'(p_i=(x_i,y_i,c_i)) = \tilde{env}'(p_i=\text{depth}(c_i,\tilde{env}'))
$$

where $\text{depth}(c_i,\tilde{env}')$ is a natural number defined structurally on the program $c$ as specified by the following table

<table>
<thead>
<tr>
<th>$c$</th>
<th>$\text{depth}(c,\tilde{env})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x:=e$</td>
<td>0</td>
</tr>
<tr>
<td>IF $b$ THEN $c_1$ ELSE $c_2$</td>
<td>max{$\text{depth}(c_1,\tilde{env}),\text{depth}(c_2,\tilde{env})$}</td>
</tr>
<tr>
<td>$c_1;c_2$</td>
<td>max{$\text{depth}(c_1,\tilde{env}),\text{depth}(c_2,\tilde{env})$}</td>
</tr>
<tr>
<td>WHILE $b$ DO $c$</td>
<td>$\text{depth}(c,\tilde{env})$</td>
</tr>
<tr>
<td>LET $x=e$ IN $c$</td>
<td>$\text{depth}(c,\tilde{env})$</td>
</tr>
<tr>
<td>PROC $p$(VAL $x$, VAR $y$) IS $c_1$ IN $c_2$</td>
<td>$\text{depth}(c_{2p},\tilde{env}(q=\text{depth}(c_1,\tilde{env})))$ where $q \not\in \text{DOM}(\tilde{env})$</td>
</tr>
<tr>
<td>CALL $p(e,y)$</td>
<td>$\tilde{env}(p)+1$</td>
</tr>
</tbody>
</table>
By induction on $k$, a natural number, we shall then prove

\[ \forall \phi \forall \psi (\phi \implies \psi) \implies \phi \land \psi. \]

The completeness result then follows since depth($\phi$) is defined for every (well-formed) program $\phi$ and reasonable environment $\psi$.

**Basis**

We shall first prove that ($\phi$) holds for $k=0$. The proof is by structural induction on the program $\phi$. The cases where $\phi$ is one of $x:=e$, $\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2$ and $c_1 ; c_2$ are very much as in the completeness proof for $\mathcal{R}$ in Section 3.4 and are therefore omitted. In the case

\[ \text{WHILE } b \text{ DO } c \]

some extra complications arise compared with the corresponding proof in Section 3.4 so we shall sketch the proof below. We also sketch the proofs for the cases $\text{LET } x:=e \text{ IN } c$ and $\text{PROC } p(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 \text{ IN } c_2$. The case $\text{CALL } p(e, y)$ cannot arise since depth($\phi$) is assumed to be zero.

**Case WHILE $b$ DO $c$:** Assume that

\[ \forall \phi \forall \psi (\phi \implies \psi) \implies \phi \land \psi. \]

holds and we shall construct a proof of the formula in $\mathcal{R}$. Let $z$ be a new variable of sort nat and define $c'$ to be the program

\[ \text{IF } z=0 \text{ THEN } \text{loop ELSE } (c_1 ; z:=z-1) \]

as in Section 3.4 and let loop be the program $\text{WHILE } \text{TRUE DO } z:=z \text{ and } z:=z-1$ be $\text{LET } z'=0 \text{ IN } ((\text{WHILE } z'+1 \text{ DO } z:=z'+1);z:=z')$ where $z'$ has sort nat and $z'=z$. Corresponding to ($\phi$) in the completeness proof for $\mathcal{R}$, case WHILE $b$ DO $c$, we have
\[
(Vu\{z\}, env) \vdash (c', s) \rightarrow_{x} s'
\]

if and only if

\[
(Vu\{z\}, env) \vdash (c, s) \rightarrow_{x} s''
\]

for some \(s''\) and \(r'\) where \(s'' \in \nu s\), \(z(s') \geq 0\) and \(z(s') = z(s) - 1\).

We now define \(P'(z)\) to express termination of \(\text{WHILE } b \text{ DO } c'\):

\[
P'(z)(s) \text{ if and only if } (Vu\{z\}, env) \vdash (\text{WHILE } b \text{ DO } c', s) \rightarrow_{x} s' \text{ for some } s' \text{ and } r
\]

where \(X\) is a vector of the variables of \(Vu\{z\}\) and \(\overline{X}\) and \(X'\) are vectors of shadow variables and new variables as before. We then have

\[
(V, env) \vdash (\text{WHILE } b \text{ DO } c, s) \rightarrow_{x} s'
\]

which implies that for some natural number \(n\)

\[
(Vu\{z\}, env) \vdash (\text{WHILE } b \text{ DO } c', s) \rightarrow_{n} s'' \text{ for some } s'' \text{ and } r'.
\]

Here (EE) follows from the expressiveness assumption. To prove (EEE) assume that

\[
(V, env) \vdash (\text{WHILE } b \text{ DO } c, s) \rightarrow_{x} s'.
\]

Since \(z \in \nu\) we can apply Lemma 4.1-3 and get

\[
(Vu\{z\}, env) \vdash (\text{WHILE } b \text{ DO } c, s) \rightarrow_{x} s''
\]

for some state \(s''\) with \(s'' \in \nu s\) and \(z(s'') = z(s')\). The proof showing that for some natural number \(n\)

\[
(Vu\{z\}, env) \vdash (\text{WHILE } b \text{ DO } c', s) \rightarrow_{n} s'''
\]

for some \(s'''\) is now straightforward by induction on proofs in \(\phi^N\).

Furthermore define the two relational formulas \(Q'\) and \(Q''\) to be

\[
G_{V, env} \{\text{WHILE } b \text{ DO } c\} \text{ and } \overline{b} \rightarrow_{G_{V, env}} [c], \text{ respectively. Let } R' \text{ and } R'' \text{ be
the time formulas $E_{V,\text{env}}$ \[\text{WHILE } b \text{ DO } c \] and $E_{V,\text{env}}$, respectively.

The rest of the proof is essentially as in the completeness proof for the case $\text{WHILE } b \text{ DO } c$, and is therefore omitted here.\\

---

**Case LET $x=e$ IN $c$:** Assume now that

1. $\forall \langle \text{LET } x=e \text{ IN } c: R \rangle (V, \text{env})$

and we shall construct a proof of the formula in $N$. Below we prove that

2. $\forall \langle \text{LET } y=e \Rightarrow R \rangle (V, \text{env})$

holds where $y$ is a variable of the same sort as $x$ satisfying $y \notin V$ and where $Q'$ and $R'$ are the two formulas $\exists y. E_{V,\text{env}}[c_x]$ and $E_{V,\text{env}}[c_x]$, respectively. The induction hypothesis then gives us a proof of

$\forall \langle \text{LET } y=e \Rightarrow R \rangle (V, \text{env})$

in $N$. Since $y \notin V$ we can apply $/\text{LET-}N$/ and get a proof of

$\forall \langle \text{LET } x=e \text{ IN } c: R \rangle (V, \text{env})$.

Below we prove that

3. $\forall \langle \text{LET } y=e \Rightarrow R \rangle (V, \text{env})$

and

4. $\forall \langle E^S(e) \Rightarrow R \rangle (V, \text{env})$

so using first $/\text{inv-}N$/ and then $/\text{cons-}N$/ we get the required proof in $N$.

To prove (2) assume that $\forall \langle \text{LET } x=e(s) \rangle (V, \text{env})$ holds for some state $s$. Then we get from (1) that for some $s'$ and $r$

5. $\forall \langle \text{LET } x=e \text{ IN } c: R \rangle (V, \text{env})$

169
The semantic rule /LET-$y^N$/ gives that for some $s'$ with $s'_0 y(s) = s'$,

$$(V\cup\{y\},env)\vdash \langle c^y_{x} s' e(s) \rangle \overset{r'}{\rightarrow} s'$$

where $e(s) + e(s') + r' = r$. From $y = e(s)$ we get $y e(s) = s$. From the expressiveness assumption it follows that $F_{G_{\cup\{y\}},env}c^y_{x}(s,s')$ and $F_{E_{\cup\{y\}},env}c^y_{x}(s,s')$. But then $F_{Q'(s,s')}$ and $F_{R'(s,r')}$ hold and (2) follows.

To prove (3) assume that $F_{PAQ'(s,s')}$ holds for some pair $(s,s')$ of states. Then $F_P(s)$ holds and from (1) we get (5). From $F_{Q'(s,s')}$ we get $F_{Q'(s e(s),s')}$ and thus for some value $v$, $F_{G_{\cup\{y\}},env}c^y_{x}(s e(s),s')$.

The expressiveness assumption then gives that

$$(V\cup\{y\},env)\vdash \langle c^y_{x} s e(s) \rangle \overset{r'}{\rightarrow} s''$$

for some $r'$ and $s''$ satisfying $s'' y(s) s'$. The semantic rule /LET-$y^N$/ now gives

$$(V,env)\vdash \langle \text{LET } x = e \text{ IN } c, s e(s) e(s') + r' \rangle \overset{s''}{\rightarrow} s''$$

and from Lemma 4.1-5 and (5) we get $s'' y(s) y(s)$. But then $s'' y(s) y(s)$ and since $\text{FV}(Q) \subseteq \cup\{y\}$ we get from (5) that $F_{Q(s,s')}$ holds. This proves (3).

To prove (4) assume that $F_{PAE^S(e)R^e_y(s,r')}$ for some pair $(s,r')$ of state and natural number. Since $F_P(s)$ holds we get from (1) that (5) holds. From $F_{E^S(e)R^e_y(s,r')}$ we get that $r = r_1 + r_2$ for some $r_1$ and $r_2$ where $F_{E^S(e)(s,r_1)}$ and $F_{R^e_y(s,r_2)}$ hold. The time expressiveness assumption together with Lemma 3.2-1 gives that $r_1 = e(s) + e(s')$. From $F_{R^e_y(s,r_2)}$ we get $F_{R^e_y(s e(s),r_2)}$ and the expressiveness condition gives

$$(V\cup\{y\},env)\vdash \langle c^y_{x} s e(s) \rangle \overset{r_2}{\rightarrow} s''$$

for some state $s''$. The semantic rule /LET-$y^N$/ now gives

$$(V,env)\vdash \langle \text{LET } x = e \text{ IN } c, s e(s) e(s') + r_2 \rangle \overset{s''}{\rightarrow} s''$$

170
and from Lemma 4.1-5 and (5) we now get \( r = e^S(s) + e(s)^+ + r_2 \), that is, \( r = r' \). From (5) we have \( \mathcal{F} \mathcal{R}(s, r) \) and (4) now follows. ///

Case PROC \( p(\text{VAL } x, \text{VAR } y) \) IS \( c_1 \) IN \( c_2 \): Assume now that

1. \( \mathcal{F} \mathcal{P}(\text{PROC } p(\text{VAL } x, \text{VAR } y) \) IS \( c_1 \) IN \( c_2 \) : \( \mathcal{R}Q/(V, \text{env}) \)

and we shall construct a proof of the formula in \( \mathcal{R}^N \). Below we shall prove that

2. \( \mathcal{F} \mathcal{P}(\text{PROC } c_{2p}, q : R \mathcal{Q}/(V, \text{env}(q=(x,y,c_1))) \)

where \( q \) is a procedure name satisfying \( q \notin \text{DOM}(\text{env}) \). The induction hypothesis can be applied and gives us a proof of the formula

\[ \mathcal{P}(c_{2p}, q : \mathcal{R}Q/(V, \text{env}(q=(x,y,c_1))) \)

in \( \mathcal{R}^N \). Using \( \text{PROC} - \mathcal{R}^N \) we then obtain the required proof of

\[ \mathcal{P}(\text{PROC } p(\text{VAL } x, \text{VAR } y) \) IS \( c_1 \) IN \( c_2 \) : \( \mathcal{R}Q/(V, \text{env}) \).

To prove (2) assume that \( \mathcal{F} \mathcal{P}(s) \) holds for some state \( s \). From (1) we then get that for some \( s' \) and \( r \)

\[ (V, \text{env}) \mathcal{P}(\text{PROC } p(\text{VAL } x, \text{VAR } y) \) IS \( c_1 \) IN \( c_2 \), \( s \xrightarrow{\mathcal{L}} s' \), \( \mathcal{F}Q(s, s') \) and \( \mathcal{F}R(s, r) \).

The semantic rule \( \text{PROC} - \mathcal{R}^N \) then gives

\[ (V, \text{env}(q=(x,y,c_1))) \mathcal{P}(c_{2p}, q : s \xrightarrow{\mathcal{L}} s') \]

and since \( \mathcal{F}Q(s, s') \) and \( \mathcal{F}R(s, r) \) hold we get that (2) follows. ///

This completes the proof of (5) in the case \( k=0 \).

THE INDUCTION STEP

Assume now that (5) holds for \( k=k_0 \) and we shall prove it for \( k=k_0+1 \).

Again we proceed by structural induction on the programs. All cases except the procedure call CALL \( p(e, y) \) are exactly as above in the
basis case and are therefore omitted here.

Case CALL p(e,y): Assume now that

(1) \(\mathcal{P}(\text{CALL } p(e,y) : R \not\rightarrow Q/(V,env))\)

and that depth(CALL p(e,y),env) = \(k_0 + 1\). We shall construct a proof of the formula in \(\mathcal{R}^N\). Below we shall prove that

(2) \(\mathcal{P}(A x = e \langle c^y, x, : R' / (V \cup \{x\}, env)\rangle\)

where env(p) = (x', y', c), x is a variable of the same sort as x' satisfying that \(x \not\in V\) and Q' and R' are the formulas \(\exists x.G \cup \{x\}, env \langle c^y, x, \rangle\) and \(E \cup \{x\}, env \langle c^y, x, \rangle\) respectively. From the definition of depth(CALL p(e,y),env) it follows that depth(c^x, y, env) \(\leq k_0\) so the induction hypothesis can be applied and we get a proof of the formula

\(\mathcal{P}(A x = e \langle c^y, x, : R' / (V \cup \{x\}, env)\rangle\)

in \(\mathcal{R}^N\). Since \(x \not\in FV(Q')\) we can apply the rule /CALL-\(\not\rightarrow\)/ and get a proof of

\(\mathcal{P}(\text{CALL } p(e,y) : E_S(e) \not\rightarrow R, x \not\rightarrow Q, x \not\rightarrow (V, env))\).

Below we prove that

(3) \(\mathcal{P}(A Q x \not\rightarrow R, x \not\rightarrow Q)\)

and

(4) \(\mathcal{P}(A Q x \not\rightarrow R, x \not\rightarrow Q)\)

so using first /inv-\(\not\rightarrow\)/ and then /cons-\(\not\rightarrow\)/ we get a proof of the required formula in \(\mathcal{R}^N\).

To prove (2) assume that \(\mathcal{P}(A x = e(s)\) holds for some state s. Then \(\mathcal{P}(s)\) holds and from (1) we get that for some s' and r

\((V, env) \not\rightarrow \langle \text{CALL } p(e,y), s \not\rightarrow s', s \not\rightarrow Q, s \not\rightarrow R(s, r)\rangle\).

The semantic rule /CALL-\(\not\rightarrow\)/ then gives that for some s' with \(s_0x(s) = s'\)
\((Vu(x), env) \downarrow \langle c^x_{y', y}, s^x_{e(s)}, s' \rangle \rightarrow s'_0\)

where \(e^x(s) + e(s)^+ r' = r\). Since \(\exists x e(x)\) we have \(s = s^e(s)\). The expressiveness assumption gives \(\exists Q'(s, s')\) and \(\exists R'(s, r')\). This proves (2).

To prove (3) assume that \(\exists Q'(x, s, s')\) for some pair \((s, s')\) of states. From \(\exists P(s)\) and (1) we then get that for some \(s''\) and \(r\)

\[
(5) \quad (V, env) \downarrow \langle \text{CALL } p(e, y), s \rangle \rightarrow s'' \quad \exists Q(s, s'') \quad \text{and } \exists R(s, r).
\]

From \(\exists Q'(x, s, s')\) we get \(\exists Q'(s^e(s), s')\) and thereby for some value \(v\)

\[
\exists C_{y, y'} x x' (s^e(s), s', v).
\]

The expressiveness assumption then gives

\[
(Vu(x), env) \downarrow \langle c^x_{y, y'}, s^x_{e(s), s'}, s' \rangle \rightarrow s'_0
\]

for some \(s'\) with \(s' = s''\). The semantic rule /CALL/ now gives

\[
(V, env) \downarrow \langle \text{CALL } p(e, y), s \rangle \rightarrow s''(s) + e(s)^+ r' \rightarrow s''.
\]

Using Lemma 4.1-5 and (5) we then get \(s'' s' x(s) \) and thus \(s'' s' x(s)\). From \(\exists F V(Q) \exists V u V\) and \(\exists Q(s, s'')\) we get \(\exists Q(s, s')\). This proves (3).

To prove (4) assume that \(\exists P A \langle s, r' \rangle E x^e(s, r')(x, r')\) for some pair \((s, r')\) of state and natural number. From \(\exists P(s)\) and (1) we get that (5) holds. From \(\exists E^s(x, r')\) we get that \(r' = r_1 + r_2\) for some \(r_1\) and \(r_2\) where \(\exists E^s(x, r_1)\) and \(\exists R(x, r_2)\) hold. The time expressiveness assumption together with Lemma 3.2-1 gives that \(r_1 = e^x(s) + e(s)^+ s''\). From \(\exists R'(e^x(s), r_2)\) we get \(\exists R'(s^e(s), r_2)\) and thereby, using the expressiveness assumption,

\[
(Vu(x), env) \downarrow \langle c^x_{y, y'}, s^x_{e(s), s'} \rangle \rightarrow r_2 + s''
\]

for some \(s''\). Then the semantic rule /CALL/ gives

\[
(V, env) \downarrow \langle \text{CALL } p(e, y), s \rangle \rightarrow s''(s) + e(s)^+ r_2 + s''.
\]

From Lemma 4.1-5 and (5) we get \(r = e^x(s) + e(s)^+ r_2\) and thereby \(r = r'\).

From \(\exists R(s, r)\) we then get the result \(\exists R(s, r')\). This proves (4).

This proves the completeness result for \(R_N\).
In order to discuss the pragmatic properties of the proof system \( X^N \) we shall in this section consider the union-find problem, also known as the equivalence problem. It can be described as follows: We have given \( n \) distinct elements denoted by the numbers 1, 2, ..., \( n \) and we have given \( n \) disjoint sets \( A_1, A_2, ..., A_n \); for the sake of simplicity we shall assume that \( A_i = \{i\} \) for \( 1 \leq i \leq n \). We have two operations

- UNION(A,B): forms the union of the two sets \( A \) and \( B \) and calls the resulting set \( A_i \)
- FIND(x): finds the set containing the element \( x \).

The problem is now to execute a sequence of \( n-1 \) UNION-operations intermixed with \( n \) FIND-operations in such a way that each operation is completed before the next one is known.

The problem has received a great deal of attention in the literature. Galler and Fischer suggested in /GaFi64/ an algorithm solving the problem based on tree structures. The idea is to arrange the elements of each set in a tree and for each element (except the root) we have a pointer to its father. The FIND-operation is accomplished by successively following the father pointers up the path from the given node of the tree. The UNION-operation is performed by letting the root of the one tree be the father of the root of the second one and mark the latter node to indicate that it is not the root of a tree any longer.

With this implementation the worst-case time complexity of the algorithm is \( \Theta(n^2) \): the time for a FIND-operation will be \( \Theta(n) \) since the maximal length of the path from the node to the root of the tree is \( n \). The time required for the UNION-operation is constant.
The algorithm can be improved by imposing a weighting rule (see for instance /AHU82/). Each time a UNION-operation is performed an attempt is made to keep the trees balanced by always attaching the tree with the smaller number of nodes to the root of that with the larger number. The height of a tree will then be at most \( \log(n) \) and thus the time for a FIND-operation becomes \( \Theta(\log(n)) \). The roots of the trees are extended with information about the cardinalities of of the sets they represent so the UNION-operation will still require constant time. The worst-case time complexity of the algorithm will therefore be \( O(n \cdot \log(n)) \).

Using the technique of path-compression the algorithm can be improved even further. The idea is that each time a FIND-operation is performed, each node on the path from the given node to the root of the tree is made a direct son of the root. The time for the subsequent FIND-operations may in this way be speeded up. Fischer proves in /Fi72/ an \( O(n \cdot \log(\log(n))) \) upper bound on the algorithm. Aho, Hopcroft and Ullman show in /AHU74/ how the analysis can be improved to prove an \( O(n \cdot \log^{*}(n)) \) upper bound where \( \log^{*}(n) \) is defined to be \( \min\{i | \log^{i}(n) \leq 1 \} \) (and \( \log^{0}(n) = n \), \( \log^{i+1}(n) = \log(\log^{i}(n)) \)). This result has been further improved by Tarjan in /Ta75/. He proves an \( O(n \cdot \alpha(n)) \) upper bound where \( \alpha(n) \) is the inverse of Ackerman's function. Since this function increases very slowly this shows that the algorithm is "almost" linear.

Although the algorithm with path-compression is quite simple and easy to understand its analysis is very complicated. We shall in this section see how the algorithm without path-compression (but with the weighting rule) can be analysed in the proof system \( \mathcal{KN} \).
As mentioned above we shall represent each set by a tree and let the nodes of the trees represent the elements of the sets. We shall use an array, father, to represent the trees and the idea is that father[i] is the father of the node representing the element i. So, for instance, the sets

\{1\}, \{2,3,4\} and \{5,6,7,8\}

may be represented by the trees

- (1)  
  (2)  
  (3)  (4)  
  (5)  
  (6)  
  (7)  (8)

which, in turn, are represented in the array father as follows

<table>
<thead>
<tr>
<th>father:</th>
<th>0 0 2 2 0 5 6 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

For the sake of simplicity we shall assume that the sets are given names being natural numbers between 1 and n. An array, set, is used to keep track of the names of the sets, their roots (in the array father) and their cardinalities. So set[i] is a record with three fields, name, root and card. More precisely, the idea is that set[i].name is the name of the set with root i, set[i].root is the root of the set named i and set[i].card is the cardinality of the set named i. If therefore the sets \{1\}, \{2,3,4\} and \{5,6,7,8\} above are named 1, 3 and 6, respectively, then the array set is as follows:

<table>
<thead>
<tr>
<th>set:</th>
<th>1 3 0 0 6 0 0 0</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 0 2 0 0 5 0 0</td>
<td>root</td>
</tr>
<tr>
<td></td>
<td>1 0 3 0 0 4 0 0</td>
<td>card</td>
</tr>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
</tbody>
</table>

176
The input to the program is a sequence of instructions of the forms \((\text{UNION},i,j)\) and \((\text{FIND},i)\) intermixed with each other. We shall not specify this in details but merely assume that we have a sort `input` and operations to read on the input.

Formally, the data type has four sorts, `nat`, `array`, `array3` and `input`. In addition to the operations of the data type of one-dimensional arrays (Example 3.1-3) we have the following operations:

- `\cdot L\cdot i\.\cdot\cdot`.name, `\cdot L\cdot i\.\cdot\cdot`.root and `\cdot L\cdot i\.\cdot\cdot`.card are function symbols of arity \((\text{array3 nat,nat})\),
- `upd-name`, `upd-root` and `upd-card` are function symbols of arity \((\text{array3 nat nat, array3})\),
- `length` is a function symbol of arity \((\text{array3 nat})\), and
- `=` is a relation symbol of arity `array3 array3`.

Furthermore we have the following operations on "input":

- `more-op` and `op-is-FIND` are relation symbols of arity `input`
- `fst-par` and `snd-par` are function symbols of arity \((\text{input},\text{nat})\),
- `tail` is a function symbol of arity \((\text{input, input})\), and
- `=` is a relation symbol of arity `input input`.

Having defined the data type we can now specify the algorithm in detail. In the following father is a variable of sort `array`, set is a variable of sort `array3`, \(i, j, k, \text{large, and small} \) are variables of sort `nat` and `inp` is a variable of sort `input`. The algorithm is as follows:
PROC union (VAL j, VAL k)
IS LET large = 0
IN LET small = 0
IN (IF set[j].card < set[k].card
    THEN (small := set[j].root; large := set[k].root)
    ELSE (small := set[k].root; large := set[j].root);
    father := update (father, small, large);
    set := upd-name (upd-name(set, large, j), small, 0);
    set := upd-root (upd-root(set, j, large), k, 0)
    set := upd-card (upd-card(set, large,
        set[small].card + set[large].card), small, 0))

IN
PROC find (VAL i)
IS WHILE ¬(father[i] = 0) DO i := father[i]

IN
WHILE more-op(inp)
DO (IF op-is-FIND(inp)
    THEN CALL find(fst-par(inp))
    ELSE CALL union(fst-par(inp), snd-par(inp));
    inp := tail(inp))

THE COMPUTATIONAL MODEL

The computational model $M$ for the data type introduced above is an extension of that considered in Example 3.1-3 for the data type of one-dimensional arrays. We have the following sets associated with the various sorts of the data type

- $\mathbb{N}_{\text{nat}}$ is the set of natural numbers,
- $\mathbb{N}_{\text{array}}$ is the set of finite sequences of natural numbers,
- $\mathbb{N}_{\text{array3}}$ is the set of finite sequences of triples of natural numbers, and
- $\mathbb{N}_{\text{input}}$ is the set of finite sequences of elements of the forms (UNION, n, n') and (FIND, n) where n and n' are natural numbers.
For \( n \in \mathcal{M}_{\text{nat}} \) we let \( n^+ = 1 \), for \( w \in \mathcal{M}_{\text{array}}, \mathcal{M}_{\text{array3}}, \) or \( \mathcal{M}_{\text{input}} \) we let \( w^+ = 0 \).

In the following let \( w = (h_1, i_1, j_1) \ldots (h_m, i_m, j_m) \) be an element of \( \mathcal{M}_{\text{array3}} \). The operations involving elements of sort \( \text{array3} \) are then interpreted as follows in \( \mathcal{M} \):

- \( w[l].\text{name} = \begin{cases} h_1 & \text{if } 1 \leq m \\ 0 & \text{otherwise} \end{cases} \)
- \( w[l].\text{root} = \begin{cases} i_1 & \text{if } 1 \leq m \\ 0 & \text{otherwise} \end{cases} \)
- \( w[l].\text{card} = \begin{cases} j_1 & \text{if } 1 \leq m \\ 0 & \text{otherwise} \end{cases} \)
- \( \text{upd-name}(w, l, h) = \begin{cases} (h_1, i_1, j_1) \ldots (h, i_1, j_1) \ldots (h_m, i_m, j_m) & \text{if } 1 \leq m \\ w & \text{otherwise} \end{cases} \)
- \( \text{upd-root}(w, l, i) = \begin{cases} (h_1, i_1, j_1) \ldots (h_1, i_1, j) \ldots (h_m, i_m, j_m) & \text{if } 1 \leq m \\ w & \text{otherwise} \end{cases} \)
- \( \text{upd-card}(w, i, j) = \begin{cases} (h_1, i_1, j_1) \ldots (h_1, i_1, j) \ldots (h_m, i_m, j_m) & \text{if } 1 \leq m \\ w & \text{otherwise} \end{cases} \)
- \( \text{length}(w) = m. \)

Finally, \( = \) is interpreted as the identity relation on \( \mathcal{M}_{\text{array3}} \). The time requirements of the operations \( \cdot[\cdot].\text{name}, \cdot[\cdot].\text{root} \) and \( \cdot[\cdot].\text{card} \) are 2 (one unit for looking up in the array and one for choosing the appropriate field). The operations \( \text{upd-name}, \text{upd-root} \) and \( \text{upd-card} \) require 3 time units each (two for selecting the correct location and one for updating it). The operation \( \text{length} \) takes one unit of time whereas the operation \( = \) requires time proportional to the length of its arguments.

The operations having arguments of sort \( \text{input} \) are interpreted as
follows in $M$

- $more-op(inp)$ holds if and only if $inp$ is not the empty string,
- $op-is-FIND(inp)$ holds if and only if $inp$ has the form $(FIND,n)inp'$ where $inp' \in \text{input}$,
  
  $\begin{cases}
  n & \text{if } inp=(FIND,n)inp' \\
  0 & \text{otherwise}
  \end{cases}$

- $fst-par(inp)$ = $\begin{cases}
  n & \text{if } inp=(UNION,n,n') \\
  0 & \text{otherwise}
  \end{cases}$

- $snd-par(inp)$ = $\begin{cases}
  n' & \text{if } inp=(UNION,n,n') \\
  0 & \text{otherwise}
  \end{cases}$

- $tail(inp)$ = $\begin{cases}
  imp' & \text{if } inp=(FIND,n)inp' \text{ or } inp=(UNION,n,n')inp' \\
  \Lambda & \text{otherwise (}\Lambda\text{ is the empty string)}
  \end{cases}$

Finally, $=$ is interpreted as the identity relation on $\text{input}$. All these operations are free (the reason being that we are not interested in the operations on input at all). This completes the specification of the computational model $M$.

**Run-time analysis in $R^N$**

Our goal will be to prove that the run-time of the union-find algorithm is bounded by $k'n\log(n)$ for some constant $k$. More precisely, we shall prove a formula of the following form

($) \text{INIT}(n) \langle \text{program:time} < k'n \log(n) \rangle \text{TRUE} / (V, ())$

where $V = \text{FV}(\text{program}) \cup \{n\}$ and $\text{INIT}(n)$ is a pure formula of the assertion language satisfying that for any state $s$

$\models \text{INIT}(n) (s)$

if and only if

$n > 1$, $\text{father}(s) = 0 \ldots 0$ ($n$ 0's), $\text{set}(s) = (1,1,1) \ldots (n,n,1)$, and $\text{inp}(s)$ is a sequence of $n-1$ triples $(\text{UNION},n'n'\ldots'n)$ and $n$ pairs $(\text{FIND},n')$ intermixed between each others (and with $\not\in n',n'\not\in n$).
That is, the data structure is initialised to represent the \( n \) sets \( \{1, \ldots, n\} \) named \( 1, \ldots, n \), respectively.

We have at least three strategies we might attempt in order to get a proof of (\$). One possibility is to proceed in a \textit{bottom-up} manner and (hopefully) end up with a \( kn \log(n) \) upper bound for some \( k \). Alternatively, we could guess some value for \( k \) and then try to verify that it is correct using a \textit{top-down} approach. A third possibility, and the one we shall use here, is to \textit{look for conditions on a term}, say \( T(n) \), that will ensure that we have a proof of the formula

\[
\text{INIT}(n) \langle \text{program:time} T(n) \rangle \text{TRUE}/(V,()).
\]

More precisely, we shall construct a "proof" for this formula in \( R^N \) without knowing exactly what term \( T(n) \) denotes. In such a "proof" we have to make some deductions about time formulas and we shall therefore impose some conditions that have to be satisfied by \( T(n) \). So, in other words, we are producing a set of \textit{verification conditions} that has to be satisfied by the term \( kn \log(n) \) (for some \( k \)) in order to turn a "proof" of (1) into a proof of (\$) in \( R^N \).

The plan for the proof

We shall first consider the main program and show that in order to prove (1) it is sufficient to prove two formulas of the form

\[
\text{(2) } P(z+1) \langle \text{op-is-FIND(inp)} \langle \text{CALL find(...)} \rangle R \rangle P(z+1) \langle Q/(VU\{z\}, env) \rangle
\]

and

\[
\text{(3) } P(z+1) \langle \text{op-is-FIND(inp)} \langle \text{CALL union(...)} \rangle R \rangle P(z+1) \langle Q/(VU\{z\}, env) \rangle.
\]

Here \( P(z) \) is the invariant of the while loop of the main program and \( Q \) is a relational formula expressing that the input has not been
changed by the procedure calls, that is, $Q$ is the formula $\text{inp} = \text{inp}$.

The time formula $R$ is not known yet; however we shall have a few conditions relating $R$ and $T(n)$ and thus ensuring that we from proofs of (2) and (3) really can get a proof of (1).

This first part of the proof is mainly performed in a top-down manner. The rest of the analysis will be bottom-up. We shall analyse the bodies of the two procedures and find time formulas $R_F$ and $R_U$ such that the formulas

$$(2') \quad P(z+1) \land \text{op-is-FIND(inp)} \langle \text{CALL find(...)} : R_F \rangle P(z+1) \land Q/(V_{\text{uf}z}, \text{env})$$

and

$$(3') \quad P(z+1) \land \text{op-is-FIND(inp)} \langle \text{CALL union(...)} : R_U \rangle P(z+1) \land Q/(V_{\text{uf}z}, \text{env})$$

are provable in $\mathcal{R}^N$.

We can now add two new conditions to the previous ones, namely $R_F \rightarrow R$ and $R_U \rightarrow R$. The analysis of the algorithm is completed by showing that the conditions can be fulfilled for $T(n)$ being $k' \cdot n \cdot \log(n)$ (for some $k$).

Before giving the details let us introduce two pure formulas

$\text{INP}(u, f)$ and $\text{PATH}(i, j, l)$. We shall not give the exact specifications as formulas of the assertion language but rather describe their intuitive meanings as this will be sufficient for the presentation of the formal proof.

- $\text{INP}(u, f)$ expresses that the value of $\text{inp}$ is a sequence of $u$ UNION-operations intermixed with $f$ FIND-operations,
- $\text{PATH}(i, j, l)$ expresses that the data structure is such that there is a path from $i$ to $\text{set}[j].\text{root}$ (in the array father) and it has length $l$. 

182
We have already introduced the formula \( \text{INIT}(n) \):

- \( \text{INIT}(n) \) expresses that the data structure represents the \( n \) sets \( \{1, \ldots, n\} \) named \( 1, \ldots, n \), respectively, and that \( n > 1 \).

**Proof for the main program**

Let us now see how to prove the formula (1) in \( \mathbb{R}^N \). It is sufficient to prove the formula

\[
(4) \quad \text{INIT}(n) \langle \text{WHILE more-op(inp)} \rangle \text{ DO } \ldots : \text{time} < T(n) \rangle \text{TRUE} / (V, \text{env})
\]

where \( \text{env} \) is the environment containing the declarations of the procedures \( \text{union} \) and \( \text{find} \). This follows by applying appropriate versions of the rule \( /\text{PROC-} \mathbb{R}^N/ \) to (4). As invariant, \( P(z) \), for the while loop we shall use the following formula

\[
\exists u. \exists f. (z = u + f \land \text{INP}(u, f)) \land (\Sigma_{j=1}^{n} \text{set}[j].\text{card} = n) \land \\
\forall i. \exists j. \exists l. (\text{PATH}(i, j, l) \land (\text{father}[i] = 0 \rightarrow l = 0)) \land \\
(\neg \text{father}[i] = 0 \rightarrow l > 0 \land \text{PATH}(\text{father}[i], j, l-1)) \land \\
\forall i. \forall j. \forall l. (\text{PATH}(i, j, l) \rightarrow l \leq \log(\text{set}[j].\text{card})).
\]

This formula expresses that there are \( z \) operations left on input, \( u \) UNION-operations and \( f \) FIND-operations. Furthermore, the sum of the cardinalities of the \( n \) sets is equal to \( n \). Each element will be on the path to the root of some set and its distance from the root will be larger than that for its predecessor on the path (if any). Finally, the formula expresses that the trees are balanced: the length of the path from a given node to the root of a tree is bounded by the logarithm of the cardinality of the set represented by the tree.

From the informal definition of the formula \( \text{INIT}(n) \) we get

\[
\neg \text{INIT}(n) \Rightarrow \exists z. P(z).
\]
If therefore the time formula $R'$ is such that

$$\forall \text{INIT}(n) \land R' \rightarrow \text{time} < T(n)$$

then we can get a proof of (4) by applying $\text{/cons-} R^N$ to a proof of the following formula

$$(4') \ \exists z. P(z) \langle \text{WHILE more-op(inp) DO ... :} R' \rangle \rightarrow \text{TRUE} / (V, \text{env})$$

In order to prove (4') we shall construct a proof of the formula

$$(5) \ P(z+1) \land \text{more-op(inp)} \langle \text{IF op-is-FIND(inp) THEN ... ELSE ... :} R' \rangle \rightarrow \ P(z) \land Q' / (V \cup \{z\}, \text{env})$$

for some time formula $R$ satisfying certain conditions. To see that this is sufficient let $Q'$ be the relational formula

$$\forall u. \forall f. \text{INP}(u, f) \rightarrow (\text{INP}(u-1, f) \lor \text{INP}(u, f-1))$$

expressing that the execution of the body of the loop will remove either a UNION- or a FIND-operation from the input. Clearly, we have

$$\forall P(z) \land \text{more-op(inp)} \land I \rightarrow \text{TRUE}$$ and $$\forall Q' \rightarrow \text{TRUE} \rightarrow \text{TRUE}. \ $$

Assuming that the time formulas $R$ and $R'$ satisfy

$$\forall P(z) \land \text{more-op(inp)} \land \text{time} = 0 \rightarrow R'$$ and

$$\forall \text{time} = 0 \rightarrow R'$$

we can apply $\text{/WHILE-} R^N$ to a proof of (5) and get a proof of (4') in $R^N$. Remember that the operations on input are free so $E(\text{more-op(inp)})$ can be chosen to be the formula $\text{time} = 0$.

Assume now that we have the proofs of the formulas (2) and (3) and we shall construct a proof of (5). Since $E(\text{op-is-FIND(inp)})$ can be chosen to be the formula $\text{time} = 0$ we get, using $\text{/IF-} R^N$ and $\text{/cons-} R^N$, a proof of the formula

$$P(z+1) \langle \text{IF op-is-FIND(inp) THEN ... ELSE ... :} R \rangle P(z+1) \land Q / (V \cup \{z\}, \text{env}).$$
From $\text{ass-}R^N$ we get a proof of since $E^{S}(\text{tail}(\text{inp}))$ can be chosen to $\text{time}=0$. We have

$$\forall u \in \text{inp:} \text{tail}(\text{inp})/\forall u \in \text{inp:} \text{tail}(\text{inp}) \Rightarrow P(z) \land Q'$$

(follows from $\forall u \in \text{inp:} \text{tail}(\text{inp}) \Rightarrow \text{INP}(u,f_1) \lor \text{INP}(u,f-1)$, that is, removing one element from the sequence inp will mean that either the number of UNION-operations or the number of FIND-operations have been reduced by one). Furthermore, $P(z+1) \land \text{more-op(inp)} \Rightarrow P(z+1)$ so using first $;/\text{cons-}R^N/$ and then $/\text{cons-}R^N/$ we get a proof of (5).

This completes the first part of the proof. We have shown that we from proofs of the formulas (2) and (3) can get a proof of the formula (1) provided that there exists a time formula $R'$ satisfying

$$\forall \text{time}=0 \Rightarrow R'$$

and

$$\forall \text{INIT}(n) \Rightarrow R' \Rightarrow \text{time} < T(n).$$

Proof for the procedure find

Let us now see how to construct a proof for the formula (2') for the procedure find. The invariant, $P'(y)$, for the while loop of the body of the procedure is chosen to be

$$P(z+1) \land \exists j. \text{PATH}(i,j,y),$$

that is, it expresses that the length of the path from the element $i$ to the root of the tree in which it occurs is $y$. Using the axiom $/\text{ass-}R^N/$ we then get a proof of

$$P'(y+1) \land \text{father}[i]=0 \land i: \text{father}[i]: \text{time}=3 \lor \forall u \in \text{inp:} \text{tail}(\text{inp}) \Rightarrow P'(y+1) \land \text{father}[i]=0 \land i: \text{father}[i]: \text{time}=3 \lor \forall u \in \text{inp:} \text{tail}(\text{inp})$$

(V=\text{inp}, y, i, env)
since $E^S(father[i])$ can be chosen to be $time=3$. We have

$$P'(y+1) \land father[i]=0 \land V_{i=z,y,i} \land i=father[i] \Rightarrow$$

$$P'(y) \land i=0 \land father[i]=i \land P(z+1) \land Q$$

because $PATH(i,j,y+1) \land i=father[i] \Rightarrow PATH(i,j,y)$ follows from the definition of $P(z)$ and $Q$ is defined to be $inp=inp$. So using $\langle inv-\rangle$ and $\langle cons-\rangle$ we get a proof of

$$P'(y+1) \land father[i]=0 \langle i=father[i] ; time=3 \rangle P'(y) \land i=0 \land father[i]=i \land P(z+1) \land Q \langle V_{i=z,y,i}, env \rangle.$$ 

The next step is now to apply the rule $\langle WHILE-\rangle$. We have

$$P'(y) \land father[i]=0 \land V_{i=z,y,i} \Rightarrow P(z+1) \land Q$$

and

$$P(P(z+1) \land i=0 \land father[i]=P(z+1) \land Q) \Rightarrow P(z+1) \land Q.$$ 

Let now the run-time of the while loop be given by the time formula $R'_F$ defined by

$$\forall i,j. PATH(i,j,l) \Rightarrow time=7'1+4.$$ 

Then we have

$$P'(y) \land father[i]=0 \land time=4 \Rightarrow R'_F$$

because $P PATH(i,j,l) \land father[i]=0 \land l=0$ follows from the definition of $P(z)$. Furthermore,

$$P(time=4) \land (time=3) \land (i=0 \land father[i]=P(z+1) \land Q) \Rightarrow R'_F \Rightarrow R'_F$$

holds because $PATH(i,j,l) \land i=0 \land father[i] \Rightarrow 1 \land 0 \land PATH(i,j,l-1)$ follows from the definition of $P(z)$. Since $E^S(father[i])=0$ can be chosen to be $time=4$ we get from $\langle WHILE-\rangle$ a proof of

$$\exists y. P'(y) \langle WHILE \rightarrow father[i]=0 \rangle \Rightarrow P(z+1) \land Q \langle V_{i=z,i}, env \rangle.$$ 

We have $P(z+1) \land op-is-FIND(inp) \land i=fst-par(inp) \Rightarrow \exists y. P'(y)$ because
\[ y \exists j. \text{PATH}(i, j, y) \text{ follows from the definition of } P(z). \text{ So using } /\text{inv-}\mathcal{K}^N/ \text{ we get a proof of } \]
\[ P(z+1) \land \text{op-is-FIND}(\text{inp}) \land i = \text{fst-par}(\text{inp}) \langle \text{WHILE } \neg \text{father}[i] = 0 \text{ DO} \ldots \rangle \\ \quad R_F \rightarrow P(z+1) \land \text{Q}/(\forall u\{z, i\}, \text{env}). \]

We shall now apply a version of the rule /\text{CALL-}\mathcal{K}^N/ \text{ where the procedure has a single call-by-value parameter. We then get a proof of } \]
\[ P(z+1) \land \text{op-is-FIND}(\text{CALL find(\text{fst-par}(\text{inp})); (time=1)} \otimes \text{fst-par}(\text{inp})) \]
\[ P(z+1) \land \text{Q}/(\forall u\{z\}, \text{env}) \]

since \( E^S(\text{fst-par}(\text{inp})) \) can be chosen to be \( \text{time}=1 \). We shall now choose \( R_F \) to be the time formula
\[ \text{time}\leq^{7'}(\log(n)+1). \]

Then we have
\[ \forall P(z+1) \land \text{op-is-FIND}(\text{inp}) \land (\text{time}=1) \otimes \text{fst-par}(\text{inp}) \rightarrow R_F \]

because we from the definition of \( P(z) \) get that \( \text{PATH(\text{fst-par}(\text{inp}), j, l) \text{ holds for some } j \text{ and } l \text{ and furthermore, that } l \leq \log(set[j].\text{card}) \) and that \( \text{set[j].card} \leq n \). Since \( \text{time=7'+4+1} \) follows from the definition of \( R_F \) we therefore have \( \text{time}\leq^{7'}(\log(n)+1) \), that is \( R_F \) holds. So using /\text{inv-}\mathcal{K}^N/ \text{ and then } /\text{cons-}\mathcal{K}^N/ \text{ we get a proof of (2')} as required.

**Proof for the procedure union**

Let us now turn to the proof of the formula (3') for the procedure union. It is straightforward but tedious to prove the following formula

\[ (3'') P(z+1) \land \text{op-is-FIND}(\text{inp}) \land j = \text{fst-par}(\text{inp}) \land k = \text{snd-par}(\text{inp}) \land \text{large}=0 \land \text{small}=0 \langle \text{IF } \text{set[j].card} \geq \text{set[k].card} \text{ THEN } ... \text{ ELSE } ... \rangle \]
\[ \text{time}=5s \rightarrow P(z+1) \land \text{Q}/(\forall u\{z, j, k, \text{large, small}\}, \text{env}). \]

We shall omit the details since they do not give further insight.
Given the proof of (3") we can now apply the rule /LET-\mathcal{N}/ twice (and then /cons-\mathcal{N}/) and we get a proof of

\[ P(z+1) \land \text{op-is-FIND(inp)} \land j=\text{fst-par}(inp) \land k=\text{snd-par}(inp) \]

\[ \langle \text{LET large}=0 \text{ IN } \ldots \text{time}=59 \rangle \langle P(z+1) \land Q)/(\forall u\in z,j,k),e\rangle. \]

We shall now use a variant of the procedure call rule /CALL-\mathcal{N}/ where the procedure has two call-by-value parameters and no call-by-variable parameters. We then get a proof of (3') for \( R_U \) being the time formula \text{time}=61.

Completion of the proof for (\$)

We have now obtained a proof of (1) for every term \( T(n) \) satisfying the following five conditions for some time formulas \( R \) and \( R' \):

- \( \exists \text{time}(\log(n)+1) \rightarrow R \),
- \( \exists \text{time}=61 \rightarrow R \),
- \( \exists P(z) \land \text{more-op}(inp) \land \text{time}=0 \rightarrow R' \),
- \( \exists R \&(Q',R') \rightarrow R' \),

and

- \( \exists \text{INIT}(n) \land R' \rightarrow \text{time}<T(n) \).

The two first formulas are satisfied for \( R \) being the time formula \text{time}=61'(\log(n)+1). We now choose \( R' \) to be the time formula

\[ \forall u,v.f.(\text{INP}(u,f) \rightarrow \text{time}<61'(u+f)'(\log(n)+1)). \]

It is easy to verify that both the third and the fourth condition now will be satisfied. Finally, for \( T(n) \) being \( 122'n'\log(n) \) we have also satisfied the last condition. This completes the proof of (\$).

\textit{Comparison}

The informal analysis of the union-find algorithm given earlier in this chapter is quite simple: Given the data structure one starts by
observing that the trees are kept balanced by the body of the union procedure (and the find procedure). Since a tree will contain at most $n$ nodes this means that the find procedure will have to search through a path of maximal length $\log(n)$ and thus its run-time requirements will be of order of magnitude $\log(n)$. The union procedure, on the other hand, will require constant time. The input to the complete algorithm consists of $n-1$ UNION-operations and $n$ FIND-operations so the run-time requirements of the algorithm will be of order of magnitude $n \log(n)$.

The important question is now whether the formal proof is a "natural formalisation" of the informal one. First we note that the overall structure of the two analyses is the same. We are in both cases looking for a term bounding the run-time of the algorithm. The two procedure bodies are analysed independently of each other (and of the main program), and the information obtained in this way is used in the analysis of the main program. In the formal proof we have presented the analysis of the main program before those of the procedures (in order to illustrate the use of "unknown" time formulas) but we could as well have reversed the order as in the informal analysis.

Let us now consider some of the details. In the informal proof we start by "observing that the trees are kept balanced by the body of the union procedure (and the find procedure)". The property that the trees are balanced is expressed by the subformula

$$\forall i. \forall j. \forall l. (\text{PATH}(i,j,l) \Rightarrow l \leq \log(\text{set}[j].\text{card}))$$

of $P(z)$. The proofs showing that $P(z)$ is kept invariant by the procedure calls formalise the "observation that the trees are kept balanced". So there is a rather close correspondence between the phrase of
the informal proof and what has actually been proved in the formal proof.

The next phrase of the informal analysis states that "since a tree will contain at most \( n \) nodes this means that the find procedure will have to search through a path of maximal length \( \log(n) \)." This corresponds essentially to the formal proof for the procedure find. The invariant \( P'(1) \) of the loop of the procedure body expresses that the path from the current node (i) to the root of the tree has length 1. We then prove that the run-time requirements of the loop satisfy the time formula \( (R'_P) \):

\[
\forall i. P'(1) \rightarrow \text{time} = 7'1 + 4
\]

that is, if the length of the path is 1 then \( 7'1 + 4 \) time units are required. Finally, we remark that the formula

\[
P(z+1) \land (\text{time} = 1) \Rightarrow (\forall i. P'(1) \rightarrow \text{time} = 7'1 + 4) \Rightarrow \text{time} \leq 7'(\log(n) + 1)
\]

of the assertion language is true. To see this we note that \( P(z+1) \) is chosen such that it both will imply that \( P'(1)_i \) holds for some \( i \) and that \( i \) is bounded by \( \log(n) \). So we get that the run-time is bounded by \( 7'(\log(n) + 1) \) and this expresses exactly that the run-time is of order of magnitude \( \log(n) \).

This substantiates the claim that it is possible to view the formal proof as a formalisation of the informal analysis: given a phrase of the informal proof we can point out the various parts of the formal proof where the corresponding formal deduction is performed.
4.6 CONCLUDING REMARKS

We have in this chapter seen how the proof system \( \mathcal{A} \) of Chapter 3 can be extended to apply to programs in a language with non-recursive procedures. The new proof system \( \mathcal{A}^N \) has been proved to be both sound and complete in (essentially) the same sense as \( \mathcal{A} \), so it has the desired theoretical properties. In order to discuss its pragmatic properties we have considered an algorithm solving the union-find problem and we have proved an \( \Theta(n \log(n)) \) upper bound for it in the proof system. Based on a comparison with the usual informal analysis we argue that we have obtained a natural formalisation of it in the proof system \( \mathcal{A}^N \).

The proof system \( \mathcal{A}^N \) of Section 4.2 specifies indirectly a proof system for total correctness of non-recursive procedure programs. The formulas of this proof system, called \( \mathcal{T}^N \), have the form \( P(c \geq Q/(V,\text{env}); \text{well-formedness and validity of these formulas are defined as expected. The axioms and rules of the proof system are the appropriate extensions of those of the proof system } \mathcal{T} \text{ of Chapter 2 together with the following three rules that easily are derived from the corresponding ones in the proof system } \mathcal{A}^N: \)

\[
/\text{LET-}\mathcal{T}^N/ \quad \frac{P(x^y = e \geq Q/(V \cup y),\text{env})}{P(\text{LET } x = e \text{ IN } c \geq Q/(V,\text{env})}
\]

where \( y \) is a variable of the same sort as \( x \) satisfying \( y \notin \text{FV}(Q) \)

\[
/\text{PROC-}\mathcal{T}^N/ \quad \frac{P(c_2 \geq Q/(V,\text{env}(q=(x,y,c))))}{P(\text{PROC } p(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 \text{ IN } c_2 \geq Q/(V,\text{env})}
\]

where \( q \) is a procedure name satisfying \( q \notin \text{DOM}(\text{env}) \)
where env(p) = (x', y', c) and x is a variable of the same sort as x' satisfying x ∈ V ∪ y' ∪ FV(Q).

The soundness and completeness results for the proof system $\mathcal{R}^N$ in the sections 4.3 and 4.4 also show that the proof system $\mathcal{J}^N$ is sound and complete (in the same sense as the proof system $\mathcal{J}$ in Chapter 2).

A comparison with proof systems in the literature shows that the three rules above are "the usual ones". A version of the rule $/\text{LET-}\mathcal{J}^N/\,$ for declaration of uninitialised variables can be found in /Ho71/ and /Ap81/ among others. The rule $/\text{PROC-}\mathcal{J}^N/\,$ is similar to a rule suggested in /Co78/ (except that we have an explicit handling of the environment) and one in /C179/. The rule $/\text{CALL-}\mathcal{J}^N/\,$ for procedure call can easily be derived from the corresponding rule suggested in for instance /Ap81/ and reflecting directly that the semantics of the construct CALL p(e, y) is equivalent to that of LET x = e IN $c^x_y$, (where env(p) = (x', y', c) and x is new):

\[ P(\text{LET } x = e \text{ IN } c^x_y, Q/(V, \text{env})) \]

\[ P(\text{CALL } p(e, y), Q/(V, \text{env})) \]

where env(p) = (x', y', c), and x has the same sort as x' and satisfies x ∈ V ∪ y' ∪ FV(Q).

In the literature one often considers dynamic scope rather than static scope. The development of this chapter can easily be modified to reflect dynamic scope rules. We shall briefly sketch how this can be accomplished. In the specification of the semantics and run-time requirements of the procedure language we obtain dynamic scope by renaming the "old" variable x rather than the "new" one in the case...
of name clashes. So we will replace the two rules /LET-$^N_1$/ and 
/CALL-$^N_1$/ of Section 4.1 by

/LET-$^N_1$/

\[
\frac{(V\cdot\{x',1\},\text{env})\vdash \langle c, s, x' \rangle \mathcal{E}(s) e(s) \xrightarrow{r} s}{(V, \text{env}) \vdash \langle \text{LET } x = e \ \text{IN } c, s \rangle \mathcal{E}(s) + e(s) + r \xrightarrow{s} s', x'(s)}
\]

where $x'$ is a variable of the same sort as $x$ satisfying $x' \not\in \mathcal{V}$.

/CALL-$^N_1$/

\[
\frac{(V\cdot\{x',1\},\text{env})\vdash \langle c', s', x' \rangle \mathcal{E}(s) e(s) \xrightarrow{r} s}{(V, \text{env}) \vdash \langle \text{CALL } (e, y), s \rangle \mathcal{E}(s) + e(s) + r \xrightarrow{s} s', x'(s')}
\]

where $\text{env}(p) = (x', y', c)$ and $x$ is a variable of the same sort as $x'$ and satisfying $x \not\in \mathcal{V}$.

As an example consider the program

PROC p(VAL y, VAR y') IS x:=x+2 IN LET x=0 IN CALL p(0,x)

Here $x$ occurs globally as well as locally. According to the rules of Section 4.1 we shall rename the local variable and (using a computational model where every operation is free) we get for instance

\[
\langle \{x\}, \text{env} \rangle \vdash \langle \text{LET } x=0 \ \text{IN } \text{CALL } p(0,x), s \rangle \mathcal{E}(s) x \xrightarrow{1} 0 0 0 3 0 0
\]

because

\[
\langle \{x,x'\}, \text{env} \rangle \vdash \langle \text{CALL } p(0,x'), s \rangle \mathcal{E}(s) x' \xrightarrow{1} 0 0 0 3 0 0
\]

/LET-$^N_1$/ which in turn follows from

\[
\langle \{x,x', y\}, \text{env} \rangle \vdash \langle x:=x+2, s \rangle \mathcal{E}(s) x' \xrightarrow{1} 0 0 0 3 0 0
\]

using /CALL-$^N_1$/ and with env being the environment ($p=(y,y', x:=x+2)$).

Using the rules above for dynamic scope we get

\[
\langle \{x\}, \text{env} \rangle \vdash \langle \text{LET } x=0 \ \text{IN } \text{CALL } p(0,x), s \rangle \mathcal{E}(s) x \xrightarrow{1} 0 0 1 0
\]

because

\[
\langle \{x,x''\}, \text{env} \rangle \vdash \langle \text{CALL } p(0,x), s \rangle \mathcal{E}(s) x'' \xrightarrow{1} 0 0 1 2 0
\]
\((/\text{LET-} \gamma_d^N/)\) which in turn follows from

\[
\{x, x'', y\}, \text{env}\} \vdash \langle x := x + 2, 1000, x'' \rangle \Rightarrow 100, \gamma_d^N
\]

using \(/\text{CALL-} \gamma_d^N/\).

The difference between the two sets of semantic rules of course gives rise to a new set of proof rules. We suggest to replace the two rules \(/\text{LET-} \gamma_d^N/\) and \(/\text{CALL-} \gamma_d^N/\) by

\[
/\text{LET-} \gamma_d^N/ \\
P^x \land x = e^y \land \langle c : R \rangle \Rightarrow \langle V(\{y\}, \text{env}) \rangle
\]

\[
P \langle \text{LET } x = e \text{ IN } c : C : E(e) \Rightarrow e_x \rangle \Rightarrow x = x / (V, \text{env})
\]

where \(y\) is a variable of the same sort as \(x\) with \(y \notin V\)

and where \(x \notin \text{FV}(Q)\).

\[
/\text{CALL-} \gamma_d^N/ \\
P^x \land x' = e^x \land \langle c : R \rangle \Rightarrow \langle V \cup \{x\}, \text{env} \rangle
\]

\[
P \langle \text{CALL } p(e, y) : C(e) \Rightarrow e_{x'} \rangle \Rightarrow x = x / (V, \text{env})
\]

where \(\text{env}(p) = (x', y', c)\), \(x\) has the same sort as \(x'\) and satisfies \(x \notin V\), and \(x' \notin \text{FV}(Q)\).

Note the substitutions of variables in these two rules are very similar to those occurring in the semantic rules above.

In Section 4.5 we distinguish between three different proof strategies for using a proof system as \(\mathcal{R}^N\) to analyse the run-time of a given algorithm. The first two are the traditional ones

I: We can analyse the algorithm in a bottom-up manner: the axioms of the proof system are used to prove properties of the basic statements of the algorithm and then the rules are used to obtain proofs of properties for larger and larger subprograms and eventually we get a proof of some property holding for the complete program.

II: We may attempt to prove some given property of the algorithm in
a top-down manner: using the rules of the proof system we show that a proof of a property for a composite program can be obtained from proofs of certain properties of its constituents and eventually we use the axioms to check that the required properties for the basic statements can be proved.

III: We can use a combination of the bottom-up and top-down approaches together with the introduction of "unknown" time formulas. Thus we construct a "proof" of some property of the algorithm together with a list of conditions on the "unknown" time formulas; for every set of time formulas fulfilling these conditions we will have a proof of the property expressed by the formula of the proof system.

Of course, the three approaches can also be mixed in various ways. The examples of the sections 3.2 and 3.5 use mainly the bottom-up approach, the example of Section 4.2 uses a combination of the top-down and the bottom-up approach and the example of Section 4.5 uses the third approach of "unknown" time formulas.

As mentioned earlier, one of our goals is to obtain natural formalisations of the traditional informal analyses of algorithms. The purpose of the informal analyses is in most cases to find some property holding for the run-time of a program and usually its exact form is not known initially. So it seems most likely that the bottom-up approach and/or the approach with the "unknown" time formulas will turn out to be successful when formalising the informal analyses. However, more experience is needed to clarify this further.
In this chapter we shall consider a variant of the programming language of the previous chapter in that we shall allow procedures to be recursively defined. We shall be interested in obtaining a proof system for analysing run-time properties of programs in this language and as in the earlier chapters we are interested in its theoretical as well as its pragmational properties.

The proof system $\mathcal{R}^N$ of Chapter 4 will not work properly for the recursive language. The most apparent reason is that the rule $\text{/PROC-}$ for procedure declaration will not be sound: if we have a declaration

PROC $p$(VAL $x$, VAR $y$) IS $c_1$ IN $c_2$ then a call of $p$ in $c_1$ will now be a recursive call of the procedure declared by this statement and this is (of course) not reflected in the rule. However, we can easily replace $\text{/PROC-}$ with another rule

$$P\\{c_2p\};R\triangleright Q/(V,\text{env}(q=(x,y,c_1^q)))$$

where $q$ is a procedure name satisfying $q \notin \text{DOM(env)}$.

The resulting proof system will be sound but not complete. Suppose, for instance that we want to prove a formula of the form

$$P\\{\text{CALL } p(e,y)\};R\triangleright Q/(V,\text{env})$$

The rule $\text{/CALL-}$ gives that it is sufficient to prove a formula of the form

$$P\\{c_y^V,x\\{x\};R\triangleright Q'/\{V\cup\{x\}\},\text{env}\}$$
where $\text{env}(p)=(x',y',c)$ and $x\downarrow V$. Now, $c$ may contain a (recursive) call of $p$ so when attempting to prove (2) we may be forced to prove a formula of the form (1) and so on. This problem is inherent from the fact that procedures may be recursively defined and let us therefore review what has been done to solve the problem for proof systems for partial and total correctness.

In /Ho71/, Hoare introduces the so-called proofs from assumptions to deal with the recursive procedure calls. In the following let $P[c]Q$ mean that the program $c$ is partially correct with respect to the pre-condition $P$ and the post-condition $Q$ (just as in Section 1.1). In the case of a single declaration of a procedure $p$ without parameters and with the body $c$, Hoare's rule for procedure call is as follows:

$$
\frac{P[\text{CALL } p]Q \vdash P[c]Q}{P[\text{CALL } p]Q}
$$

Intuitively, the rule says that if the property $P[c]Q$ can be proved from the assumption that $P[\text{CALL } p]Q$ holds for all the (recursive) calls of $p$ in $c$ then we can conclude that $P[\text{CALL } p]Q$ does indeed hold for all calls of $p$.

However, the resulting proof system is not a proof system in the usual sense of first order logic. The idea is therefore to formalise the notion "proofs from assumptions" and simply let the "assumptions" be part of the formulas of the proof system. A proof system based on these ideas is given by for instance deBakker in /dB79/ and by Harel, Pnueli and Stavi in /HPS77/.

The formulas of such a proof system will have the form $\phi_1 \land \ldots \land \phi_m \Rightarrow \phi$
where $\phi_1, \ldots, \phi_m$ and $\phi$ are the usual partial correctness formulas, that is, they have the form $P[C]Q$. The idea is that $\phi_1 & \ldots & \phi_m$ is a possible empty list of assumptions and $\phi$ is the conclusion. Using this notation Hoare's rule above can be replaced by a rule of the form

$$
\phi_1 & \ldots & \phi_m & P[CALL p]Q \vdash P[C]Q
$$

The remaining axioms and rules of the usual proof system for partial correctness can easily be extended to record these "lists of assumptions".

Let us now briefly discuss a proof system for total correctness of recursive procedure programs. Sokolowski suggests in /So77b/ a rule of (essentially) the form

$$
\neg P(0), P(z) \langle CALL p \rangle Q \vdash P(z+1) \langle C \rangle Q
$$

where $z$ ranges over the natural numbers.

The idea is that $z$ bounds the depth of the recursion so $P(z) \langle CALL p \rangle Q$ can be interpreted as saying that if initially $P(z)$ holds then the execution of CALL $p$ will terminate in a state satisfying $Q$ and at any moment at most $z$ calls of $p$ will be active. The assumption $\neg P(0)$ ensures that this holds for $z=0$ and the assumption $P(z) \langle CALL p \rangle Q \vdash P(z+1) \langle C \rangle Q$ says that if the interpretation is correct for $z$ then it will also be correct for $z+1$.

A formalisation of the "proofs from assumptions" similar to that sketched above for partial correctness proof systems but for the total correctness proof system will be presented in Section 5.2.

So far we have discussed some of the problems encountered when
constructing a proof system for (partial or total) correctness of recursive procedures. However, also the run-time analysis of recursive programs tend to be more complicated and usually involve construction and solution of recurrence relations. Aho, Hopcroft and Ullman explain it as follows in /AHU82 p24/:

"If there are recursive procedures we cannot find an ordering of all procedures so that each calls only previously evaluated procedures. What we must now do is associate with each recursive procedure an unknown time function \( T(n) \), where \( n \) measures the size of the arguments to the procedure. We can then get a recurrence for \( T(n) \), that is, an equation for \( T(n) \) in terms of \( T(k) \) for various values of \( k \)."

The informal analysis of the merge sort algorithm presented in Section 1.3 uses this idea. When discussing the pragmatic value of the proof system for analysing run-time properties of recursive procedures it is important to study to which extent we obtain the usual recurrence relations. This will be investigated in Section 5.5 where we consider a couple of worked examples.

Finally, let us give an overview of the rest of this chapter. In Section 5.1 we modify the semantics and run-time requirements of the procedure language of Chapter 4 to reflect that procedures now might be recursively defined. For the sake of simplicity we shall restrict ourselves to the case where only call-by-value parameters are allowed. The proof system \( \mathcal{R} \) for run-time analysis of programs in this language is presented in Section 5.2 and its soundness and completeness properties are investigated in the sections 5.3 and 5.4, respectively. In Section 5.5 we show how the run-time of the merge sort algorithm can be analysed in the proof system \( \mathcal{R} \). Finally, in Section 5.6 we shall consider the underlying proof system for total correctness. This proof system, of course, applies to programs in a language with nested declarations of recursive procedures whereas in the literature
one usually restrict ones attention to programs with either a single recursive procedure or a set of mutual recursive procedures.

5.1 THE PROCEDURE LANGUAGE

As mentioned we shall consider a variant of the language of the previous chapter where procedures are allowed to be recursively defined but for the sake of simplicity they are only allowed to have a single call-by-value parameter. More precisely, the syntax of the language we shall consider is as follows:

- \( x := e \) is a procedure program,
- if \( c \) and \( c' \) are procedure programs then so are IF \( b \) THEN \( c \) ELSE \( c' \), \( c; c' \), WHILE \( b \) DO \( c \), LET \( x := e \) IN \( c \) and PROC \( p(VAL \ x) \) IS \( c \) IN \( c' \),
- CALL \( p(e) \) is a procedure program.

As in Section 4.1 we shall let \( FV(c) \) and \( FP(c) \) denote the set of variables occurring free in the program \( c \) and the set of procedure names occurring free in \( c \), respectively. The definitions of the two sets are essentially as in Section 4.1; however, note that

\[
FP(\text{PROC } p(VAL \ x) \text{ IS } c \text{ IN } c') = (FP(c) \cup FP(c')) - \{p\}
\]

reflecting that procedures are recursively defined.

An environment \( env \) is defined as in Section 4.1 except that it now associates a closure of the form \((x, c)\) with each procedure name of its domain and thus reflects that procedures only have one parameter.

The well-formedness condition imposed on the procedure programs in this chapter is slightly different from that of the previous one because we allow recursion. This is reflected in the following rule

200
defining the predicate $WF(env, c)$ in the case of procedure declaration:

$$WF(env(p=(x,c)), c), \ WF(env(p=(x,c)), c')$$

$$WF(env, PROC \ p(\text{VAL} \ x) \ IS \ c \ IN \ c')$$

The remaining axioms and rules defining the predicate $WF(c, env)$ are essentially as in Section 4.1 and are therefore omitted here.

**SEMANTICS AND RUN-TIME REQUIREMENTS**

The semantics and run-time requirements of the recursive procedure language can be described by a slightly modified version of the axiomatic system $\mathcal{S}^N$ presented in Section 4.1. However, for the development later in this chapter it is very important to keep track of the depth of the recursive calls of the various procedures at any point of time. Therefore we shall extend the relation $\langle V, env \rangle \rightarrow \langle c, s \rangle \xrightarrow{\mathcal{L}} s'$ of the previous formal system $\mathcal{S}^N$ with a new component recording this information. The component will be a mapping $d$ that to each procedure name $p$ of $\text{DOM}(env)$ associates a natural number denoted $d(p)$ which is the maximal depth of recursive calls of $p$ that has occurred in the computation specified by the formula $\langle V, env \rangle \rightarrow \langle c, s \rangle \xrightarrow{\mathcal{L}} s'$. The new relation will be written

$$\langle V, env \rangle \rightarrow \langle c, s \rangle \xrightarrow{\mathcal{L}, d} s'.$$

The mapping $d$ will be called a depth counter or simply a counter. We shall write $d(p=n)$ for the counter that is as $d$ except that the procedure name $p$ has associated the value $n$.

The set $\mathcal{S}^R$ of axioms and rules defining the new relation is obtained as a straightforward modification of the previous set $\mathcal{S}^N$ given in Section 4.1. It is as follows:
Semantics and run-time of recursive procedure programs: $g^R$

/ass-$g^R$/

\[
(V, env) \vdash \langle x := e, s \rangle \xrightarrow{e(s) + e(s)}^d \langle e(s) \rangle_x
\]

where $d(p) = 0$ for $p \in \text{DOM}(env)$

/IF-$g^R$/

\[
\begin{align*}
(V, env) & \vdash \langle b, s \rangle \xrightarrow{b(s)}^d \langle s' \rangle \\
(V, env) & \vdash \langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s \rangle \xrightarrow{b''(s) + r}^d \langle s' \rangle
\end{align*}
\]

/;-$g^R$/

\[
(V, env) \vdash \langle c_1, s \rangle \xrightarrow{r}^d \langle s' \rangle, \quad (V, env) \vdash \langle c_2, s' \rangle \xrightarrow{r'}^d \langle s'' \rangle
\]

\[
(V, env) \vdash \langle c_1; c_2, s \rangle \xrightarrow{r + r'}^d \langle s'' \rangle
\]

where $d''(p) = \max\{d(p), d'(p)\}$ for $p \in \text{DOM}(env)$

/WHILE-$g^R$/

\[
(V, env) \vdash \langle b, s \rangle \xrightarrow{b(s)}^d \langle s' \rangle, \quad (V, env) \vdash \langle \text{WHILE } b \text{ DO } c, s \rangle \xrightarrow{r''}^d \langle s'' \rangle
\]

\[
(V, env) \vdash \langle \text{WHILE } b \text{ DO } c, s \rangle \xrightarrow{b''(s) + r''}^d \langle s'' \rangle
\]

where $d''(p) = \max\{d(p), d'(p)\}$ for $p \in \text{DOM}(env)$

/LET-$g^R$/

\[
(V, env) \vdash \langle x := e, s \rangle \xrightarrow{e(s) + e(s) + r}^d \langle s' \rangle
\]

where $x'$ has the same sort as $x$ and satisfies $x' \notin V$

/PROC-$g^R$/

\[
(V, env) \vdash \langle \text{PROC } p(\text{VAL } x) \text{ IS } c_1 \text{ IN } c_2, s \rangle \xrightarrow{r}^d \langle s' \rangle
\]

where $q$ is a procedure name satisfying $q \in \text{DOM}(env)$ and $d'(p') = d(p')$ for $p \in \text{DOM}(env)$ and $d'(q)$ is undefined

(cont.)
Semantics and run-time of recursive procedure programs: \( \mathcal{Y}^R \) (cont.)

\[
\begin{align*}
& (V,env) \vdash \langle c, x^t, s_x \rangle \xrightarrow{d} s' \\
& (V,env) \vdash \langle \text{CALL } p(e), s \rangle \xrightarrow{e(s)+e(s)+r} s',
\end{align*}
\]

where \( \text{env}(p)=(x,c) \), \( x' \) is a variable of the same sort as \( x \) satisfying \( x' \notin V \) and \( d'=d(p=d(p)+1) \).

Properties of (recursive) procedure programs

In the rest of this section we shall mention some results about the semantics and run-time requirements of the recursive procedure programs that will be needed in the later proofs. However, first we shall define \( FP^*(c,env) \) to be the set of procedure names from \( \text{DOM}(env) \) that may be called during an execution of the program \( c \) in the environment \( env \). Formally, \( FP^*(c,env) \) is defined to be the set \( FP^k(c,env) \) where \( k \) is the cardinality of the set \( \text{DOM}(env) \) and

\[
FP_0(c,env)=FP(c),
FP_{i+1}(c,env)=\bigcup \{FP(c')|\text{env}(p')=(x,c') \text{ for some } p' \in FP_i(c,env)\}.
\]

Note that if \( env \) is a reasonable environment then so is \( env|FP^*(c,env) \), the restriction of \( env \) to the domain \( FP^*(c,env) \).

Corresponding to the two lemmas 4.1-1 and 4.1-2 of Chapter 4 (and the lemmas 3.1-1 and 3.1-2, and 2.1-1 and 2.1-2 of the chapters 2 and 3, respectively) we have

Lemma 5.1-1: If \( (V,env) \vdash \langle c, s \rangle \xrightarrow{d} s', V' \cap (FV(c) \cup FV(env|FP^*(c,env))) = \emptyset \) and \( V' \subseteq V \), then \( s \in V_V, s' \).

Lemma 5.1-2: If \( (V',env) \vdash \langle c, s \rangle \xrightarrow{d} s', V' \cap (FV(c) \cup FV(env|FP^*(c,env))) = \emptyset \) and \( s \in V_0, s' \) then \( (V,env) \vdash \langle c, s \rangle \xrightarrow{d} s' \) where \( s' \notin V \) and the two proofs \( \mathcal{Y}^R \) have the same length.
Both lemmas can be proved by induction on the length of the proofs in $\mathcal{R}$. The proofs are straightforward so we omit the details.

In the proof of the soundness result in Section 5.3 we shall need results that tells us how to extend the $(V,env)$-part of a formula $(V,env)\vdash \langle c,s \rangle^r_d s'$ to contain further variables or procedure declarations. More precisely, we shall be interested in how the components $s'$, $d$ and $r$ are affected. We have the following two results:

Lemma 5.1-3: If $(V,env)\vdash \langle c,s \rangle^r_d s'$ and $\forall V'=\emptyset$ then $(V\forall ',env)\vdash \langle c,s \rangle^r_d s''$ where $s''^{\forall} s'$ and $s''^V s$ and the two proofs in $\mathcal{R}$ have the same length.

Lemma 5.1-4: If $(V,env)\vdash \langle c,s \rangle^r_d s'$, $q \notin \text{DOM}(env)$, $\text{FV}(c') \subseteq V \cup \{x\}$ and $\text{FP}(c') \subseteq \text{DOM}(env) \cup \{q\}$ then $(V,env(q=(x,c')))\vdash \langle c,s \rangle^r_d s'$ where $d'=d(q=0)$.

Lemma 5.1-3 corresponds very closely to Lemma 4.1-3 in Chapter 4 and is proved in essentially the same way. The lemma follows from a result about renaming of variables:

Lemma 5.1-5: Assume that $(V,env)\vdash \langle c,s_{x}^{y(s)} \rangle^r_d s'$ where $x$ and $y$ have the same sort, $x \in V$ but $y \notin V$. Then $(V\forall y,env_{x}^{y})\vdash \langle c^{y},s \rangle^r_d s''$ where $s''^{\forall} x(s) x(s')$ and the two proofs in $\mathcal{R}$ have the same length.

The lemma corresponds to Lemma 4.1-5 and its proof is a straightforward modification of that for Lemma 4.1-5 as well. Remember that $\text{env}_{x}^{y}$ is the environment that is as env except that all free occurrences of $x$ in the procedure bodies are replaced by $y$. Intuitively, the renaming of $x$ to $y$ in the environment is needed because we implement static scope by dynamic scope in the semantics.
Using Lemma 5.1-5 we can now obtain a proof of Lemma 5.1-3 by a straightforward modification of the proof of Lemma 4.1-4. We omit the details.

This proves the first of the results about the extension of the \((V,env)-\)part of a formula \((V,env)\vdash_{d}^{\mathcal{L}}_{\Delta} s'\). In order to prove the second of the results, Lemma 5.1-4, we shall need a result that allows us to rename procedures:

**Lemma 5.1-6:** Assume that \((V,env)\vdash_{p}^{q} c, s \vdash s'\), \(p \in \text{DOM}(env)\) and \(q \in \text{DOM}(env)\).

Then \((V,env)^{q}_{p} \vdash_{p'}^{d'} c, s \vdash s'\) where \(d'(p') = d(p')\) for \(p' \in \text{DOM}(env) - \{p\}\) and \(d'(q) = d(p)\). Furthermore, the two proofs in \(\mathcal{F}^R\) have the same length.

This result corresponds to that of Lemma 4.1-6 in Chapter 4 and its proof is a straightforward modification of that of Lemma 4.1-6 as well. Remember that \(\text{env}^{q}_{p}\) is the environment that is as \(\text{env}\) except that the name of the procedure \(p\) is changed to \(q\) and all calls of \(p\) occurring in the procedure bodies of \(\text{env}\) are replaced by calls of \(q\). The proof of Lemma 5.1-4 is given in Appendix C.

In the proofs of the soundness and completeness results of the sections 5.3 and 5.4, respectively, we shall need results that allow us to replace one actual (call-by-value) parameter with another one in certain situations. We have

**Lemma 5.1-7:** Assume that \(a \in V \cap \text{FV}(\text{env})\) and \(\text{FV}(e) \subseteq V\).

If \((V,env)\vdash \text{CALL} p(a), s_{a}^{e(s)} \vdash_{d}^{s} s'\) then \(r = a\left(s_{a}^{e(s)} + a(s_{a}^{e(s)}) + r'\right)\) for some \(r'\) and \((V,env)\vdash \text{CALL} p(e), s_{e}^{e(s)+e(s)} \vdash_{d}^{s''} s''\) for some state \(s''\) satisfying \(s'' =_{V \cup \{a\}} s'\).
Lemma 5.1-8: Assume that \( a \in V-FV(\text{env}) \). If

\[(V, env) \vdash \langle \text{CALL } p(e), s \rangle \xrightarrow{e(s)} s' \text{ then } r = e^+(s) + e(s) + r' \text{ for some } r' \text{ and}
\]

\[(V, env) \vdash \langle \text{CALL } p(a), s_a^e(s) \rangle \xrightarrow{r''+r'} d_{s''} \text{ for } r'' = a^+(s_a^e(s)) + a(s_a^e(s)) + \text{ and}
\]

for some \( s'' \) satisfying \( s'' \in V-\text{env} \)

The proofs of these results are sketched in Appendix C.

Finally, in the completeness proof in Section 5.4 we shall need the following lemma expressing that the procedure language is deterministic:

Lemma 5.1-9: If \((V, env) \vdash \langle c, s \rangle \xrightarrow{d} s'\) and \((V, env) \vdash \langle c, s \rangle \xrightarrow{d} s''\) then \( s' = s'' \), \( r = r' \) and \( d = d' \).

The lemma can be proved by induction on the length of the proofs in \( \mathcal{S} \). Since the details essentially are as in the proof of Lemma 4.1-5 they are omitted here.

5.2 THE PROOF SYSTEM \( \mathcal{R} \)

The formulas of the proof system \( \mathcal{R} \) for proving run-time properties of programs in the recursive procedure language have the form

\[ P_1 \langle c_1:R_1 \rangle Q_1 \land \ldots \land P_k \langle c_k:R_k \rangle Q_k 
\]

(where \( k \geq 0 \)). Here \( c_0, c_1, \ldots, c_k \) are procedure programs (as defined in Section 5.1) and \( P_i, Q_i \) and \( R_i \) (for \( 0 \leq i \leq k \)) are pure formulas, relational and time formulas, respectively, of the assertion language. As in the proof system \( \mathcal{R} \) of Chapter 4, \( V \) is a finite set of program variables and \( \text{env} \) is an environment. The role of the pair \((V, env)\) is exactly as in the system \( \mathcal{R} \) and it applies to each of the formulas \( P_i \langle c_i:R_i \rangle Q_i \) (0 \( \leq i \leq k \)) rather than just one of them. The component \( L \) is a finite set of pairs of procedure names from \( \text{DOM}(\text{env}) \) and program
variables of sort $\text{nat}$ from $V$. Intuitively, the idea is that if $(p,z)$ is in $L$ then the value of $z$ bounds the depth of the recursive calls of $p$. This gives a variable as $z$ a special status and this is reflected both when we below impose a well-formedness condition on the formulas and when we later define their validity.

A formula $P(c:R)Q/(V,env)$ is well-formed relative to $L$ if the following conditions are satisfied:

- $\text{VAR}(L) \subseteq V$, $\text{PROC}(L) \subseteq \text{DOM}(env)$ and if $(p,z) \in L$ and $(p,z') \in L$ then $z \geq z'$ and if furthermore $(p',z) \in L$ then $p = p'$,
- $\text{FV}(env) \subseteq V - \text{VAR}(L)$ and $\text{FP}(env) \subseteq \text{DOM}(env)$,
- $\text{FV}(c) \subseteq V - \text{VAR}(L)$ and $\text{FP}(c) \subseteq \text{DOM}(env)$,
- $\text{FP}(P) \subseteq V$, $\text{FV}(Q) \subseteq V \cup \text{time}$.

Here $\text{VAR}(L) = \{z \mid (p,z) \in L\}$ and $\text{PROC}(L) = \{p \mid (p,z) \in L\}$. Note that the well-formedness condition restricts the use of the variables of $L$ such that they only can occur in the formulas of the assertion language ($P$, $Q$ and $R$) and not in the programs ($c$).

As mentioned earlier the idea is that $P_1 \langle c_1:R_1 \rangle Q_1 \& \ldots \& P_k \langle c_k:R_k \rangle Q_k$ is a list of assumptions and we shall impose the restriction that if one of the assumptions may involve calls of a procedure $p$ then $p$ must be in $\text{PROC}(L)$, that is, it must be defined in $\text{env} \cap \text{PROC}(L)$, the restriction of the environment $\text{env}$ to the domain $\text{PROC}(L)$. We say that the formula $P_1 \langle c_1:R_1 \rangle Q_1 \& \ldots \& P_k \langle c_k:R_k \rangle Q_k \Rightarrow P(c:R)Q/(V,env,L)$ is well-formed if

- $P_1 \langle c_1:R_1 \rangle Q_1/(V,env,\text{PROC}(L))$ is well-formed relative to $L$ for $1 \leq i \leq k$, and
- $P(c:R)Q/(V,env)$ is well-formed relative to $L$.

Assume now that we have given a computational model for the data
type. The validity definition for the formulas is slightly complicated because of the association of procedure names and variables in the set $L$. As mentioned the idea is that if $(p, z)$ is in $L$ then the value of $z$ bounds the depth of the recursive calls of $p$. The exact depth of recursive calls is given by the depth counter $d$ of the formulas $(V, \text{env})\langle c, s \rangle^d_{L} s'$ specified in Section 5.1. Given a state $s$ we now define $s^L_d$ to mean that for every pair $(p, z)$ of $L$ $z(s) \geq d(p)$. Similarly, we define $s_L^L_d$ and $s = L_d$.

Given a counter $d$ and a set $L$ we define the $d:L$-validity of a formula $P\langle c : R \rangle Q/(V, \text{env})$ that is well-formed relative to $L$ to mean that

for every state $s$ if $s_L^L_d$ and $\forall P(s)$ hold then for some state $s'$, natural number $r$ and depth counter $d'$

$$(V, \text{env})\langle c, s \rangle^L_d s', \forall Q(s, s'), \forall R(s, r) \text{ and } s_L^L_d'.$$

We shall write $d:L \forall P\langle c : R \rangle Q/(V, \text{env})$ for the $d:L$-validity of the formula $P\langle c : R \rangle Q/(V, \text{env})$.

The idea is that $d$ bounds the depth of the recursive calls of the procedures of $\text{PROC}(L)$. The conditions $s_L^L_d$ and $s_L^L_d'$ ensure that this is the case since $d(p) \geq z(s) \geq d'(p)$ will hold for every $(p, z)$ of $L$. Note that the conditions also ensure that the values of the variables of $\text{VAR}(L)$ in $s$ are upper bounds on the recursion depth of the various procedures.

The globality of $d$ in the $d:L$-validity definition is important when defining the validity of the general formulas because it allows us to bound the recursion depth of the procedures globally. A well-formed formula $P_1\langle c_1 : R \rangle Q_1 \& \ldots \& P_k\langle c_k : R \rangle Q_k \Rightarrow P\langle c : R \rangle Q/(V, \text{env}, L)$ is valid if for every depth counter $d$ (with domain $\text{DOM}(\text{env})$) we have
if \( d : \models P_1 < c_1 : R_1 > Q_1 / (V, env) \) holds for \( 1 \leq i \leq k \) then \( d : \models P < c : R > Q / (V, env) \) holds as well.

We shall write \( \models P_1 < c_1 : R_1 > Q_1 \ldots \ldots P_k < c_k : R_k > Q_k \) for this validity.

The definition captures the idea that \( P_1 < c_1 : R_1 > Q_1 \ldots \ldots P_k < c_k : R_k > Q_k \) is a list of assumptions and when they are fulfilled for some counter \( d \) then so is the formula \( P < c : R > Q \). In the following we shall write \( \models \) for a list of the form \( P_1 < c_1 : R_1 > Q_1 \ldots \ldots P_k < c_k : R_k > Q_k \) (\( k \geq 0 \)). A member of the list will be denoted \( \phi \), thus \( \phi \) is a formula of the form \( P < c : R > Q \). Finally, we shall write \( \emptyset \) for the empty list \( \phi \).

**Axioms and Rules**

Most of the axioms and rules of the proof system \( \mathcal{R} \) are simple extensions of those of the previous proof system \( \mathcal{R}^N \) obtained by "adding" lists of assumptions and procedure name/variable associations to the various formulas. However, we shall need a couple of new axioms and rules in order to cope properly with the procedure calls. We shall list the complete set of axioms and rules below and they will be explained afterwards.

The proof system \( \mathcal{R} \)

\[
\text{ass-} \mathcal{R}^R / \quad \Phi \models P(x := e : E) I_v - \{ x \}^x = e / (V, env, L)
\]

\[
\text{IF-} \mathcal{R}^R / \quad \Phi \models P < \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 > Q / (V, env, L), \quad \Phi \models P < c_2 : R > Q / (V, env, L)
\]

\[
\text{i-} \mathcal{R}^R / \quad \Phi \models P < c_1 : R > Q / (V, env, L), \quad \Phi \models P < c_2 : R > Q / (V, env, L)
\]

\[
\Phi \models P < c_1 : R > Q_1 / (V, env, L), \quad \Phi \models P < c_2 : R > Q_2 / (V, env, L)
\]

(cont.)
The proof system $R^K$ (cont.)

\[
\Phi \Rightarrow P(z+1)\forall b: c: R \Rightarrow P(z) \forall Q'(V, x, L), P(0) \Rightarrow \neg b,
\]

\[
P(z) \forall b \alpha \forall V \Rightarrow Q, \quad Q' \forall Q \Rightarrow Q,
\]

\[
\begin{align*}
\Phi & \Rightarrow \exists z. P(z) \langle \text{WHILE } b \text{ DO } c: R \Rightarrow Q \rangle (V, x, L)
\end{align*}
\]

where $z$ is a variable of sort $\text{nat}$ satisfying $z \notin V$

\[
\begin{align*}
\Phi & \Rightarrow P(\forall x = e \rightarrow E^S (e) \forall R \Rightarrow Q \rangle (V, x, L))
\end{align*}
\]

where $x'$ is a variable of the same sort as $x$ satisfying $x' \notin V \cup \text{FV}(Q)$

\[
\begin{align*}
\Phi & \Rightarrow P(c \forall R \Rightarrow Q \rangle (V, x, L))
\end{align*}
\]

\[
\Phi \Rightarrow P(\forall Q' \Rightarrow Q \rangle (V, x, L))
\]

where $q$ is a procedure name satisfying $q \notin \text{DOM}(env)$

\[
\begin{align*}
\Phi & \Rightarrow P(z) \langle \text{CALL } p(a); E^S (a) \forall R \Rightarrow Q \rangle (V, x, L)
\end{align*}
\]

where $\text{env}(p) = (x, c), (p, z) \notin L$ and $z$ is a variable of sort $\text{nat}$ satisfying $z \notin \text{FV}(R)$ and $z \notin \text{FV}(Q)$; $a$ is a variable of the same sort as $x$ satisfying $a \notin V \cup \text{FV}(Q)$

\[
\begin{align*}
\Phi & \Rightarrow P(c: R \Rightarrow Q \rangle (V, x, L))
\end{align*}
\]

where $P(c: R \Rightarrow Q)$ is in $\Phi$

\[
\begin{align*}
\Phi & \Rightarrow P(\forall Q' \Rightarrow Q \rangle (V, x, L))
\end{align*}
\]

where $a \in V - (\text{FV}(\text{env} \cup \text{PROC}(L))) \cup \text{FV}(Q))$ (cont.)

210
The proof system $\mathcal{R}$ (cont.)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>/elim-$\mathcal{R}$/</td>
<td>$\phi \Rightarrow P\langle c:R\rangle Q/(V_0z_1,\ldots,z_k,env,L')$</td>
<td>$\phi \Rightarrow \exists z_1\ldots\exists z_k P\langle c:R\rangle Q/(V,env,L)$</td>
</tr>
<tr>
<td></td>
<td>where $L'\cap L=\emptyset$, $\text{VAR}(L')\cap V=\emptyset$ and $\text{VAR}(L')={z_1,\ldots,z_k}$</td>
<td></td>
</tr>
<tr>
<td>/ext-$\mathcal{R}$/</td>
<td>$\phi \Rightarrow P\langle \text{CALL } p(e):R\rangle Q/(V,env,L)$</td>
<td>$\phi \Rightarrow P\langle \text{CALL } p(e):R\rangle Q/\text{AI}_V/(V,env,L)$</td>
</tr>
<tr>
<td></td>
<td>where $V\subseteq V(\text{FV(env)PROC(L)})$</td>
<td></td>
</tr>
<tr>
<td>/cons-$\mathcal{R}$/</td>
<td>$P\Rightarrow P'$, $\phi \Rightarrow P\langle c:R\rangle Q'//(V,env,L)$, $Q'\Rightarrow Q$, $R\Rightarrow R$</td>
<td>$\phi \Rightarrow P\langle c:R\rangle Q/(V,env,L)$</td>
</tr>
<tr>
<td>/inv-$\mathcal{R}$/</td>
<td>$\phi \Rightarrow P\langle c:R\rangle Q/(V,env,L)$</td>
<td>$\phi \Rightarrow P\langle c: \text{PAR}\rangle PAQ/(V,env,L)$</td>
</tr>
</tbody>
</table>

We shall write $\mathcal{R}^R\vdash \phi \Rightarrow P\langle c:R\rangle Q/(V,env,L)$ if the formula $\phi \Rightarrow P\langle c:R\rangle Q/(V,env,L)$ is provable using the axioms and rules above with the restriction that an axiom or rule only can be applied if it yields a well-formed formula and if the formulas from the assertion language that are used all are true in the given computational model.

The axiom /ass-$\mathcal{R}$/ and the rules /IF-$\mathcal{R}$/, /-/$\mathcal{R}$/, /WHILE-$\mathcal{R}$/ and /LET-$\mathcal{R}$/ are straightforward extensions of the corresponding ones in the proof system $\mathcal{R}^N$ (see Section 4.2). The rule /PROC-$\mathcal{R}$/ is the obvious extension of the previous rule /PROC-$\mathcal{R}^N$/ reflecting that procedures are recursively defined.

The previous rule /CALL-$\mathcal{R}^N$/ for procedure call have been replaced by four new rules, /CALL-$\mathcal{R}$/, /par-$\mathcal{R}$/, /elim-$\mathcal{R}$/ and /ext-$\mathcal{R}$/ and one axiom, /sel-$\mathcal{R}$/. The rules /CALL-$\mathcal{R}$/ and /elim-$\mathcal{R}$/ are usually used in situations where the assumption list does not contain informa-
tion about the procedure call at hand whereas the axiom /sel-\text{R}^R/ and
the rules /par-\text{R}^R/ and /ext-\text{R}^R/ are used when the assumption list
contains the required information.

The hypothesis of the rule /CALL-\text{R}^R/ extends the list of assumptions
with some property of a call of the procedure p and states that a
similar property can be obtained for the procedure body. The variable
a represents the value of the call-by-value parameter and it is also
used when replacing the formal parameter x of the procedure with a new
variable. We could as well have introduced a new variable x' and re-
placed the hypothesis of the rule by

\[ \text{F} \& \text{P}(z) < \text{CALL} p(a); E_s^S(a) \@ R \Rightarrow P(z+1) \wedge x' = a \langle c \times x': R \rangle Q / \{ V \{ a, x' \}, \text{env}, L \}. \]

This would be more in the style of the rule /CALL-\text{R}^N/ of Section 4.2
but the distinction between x' and a is unnecessary. The formula R
expresses a property of the run-time of the procedure body. It does not
take the run-time for evaluating the parameter into account. The
formula E_s^S(a) \@ R expresses a property of the run-time for a call of p
with the actual parameter a and when we in the conclusion have the
actual parameter e we get that E_s^S(e) \@ R^e_a holds for the run-time.

The variable z counts the maximal depth of recursive calls of the
procedure p and this association of p with z is reflected by the
assumption of /CALL-\text{R}^R/ that (p,z) is in L. The rule /elim-\text{R}^R/ is
used to add such pairs to the set L.

The axiom /sel-\text{R}^R/ should be straightforward to understand. The
parameter substitution performed in the rule /par-\text{R}^R/ is very similar
to that performed in the rule /CALL-\text{R}^R/ and should not need further
explanation. Also the rule /ext-\text{R}^R/ should be straightforward to
understand: it simply records that certain variables cannot be changed
by executing the procedure call.

Finally, we have the two general rules /cons-R/ and /inv-R/ and they are the straightforward extensions of the corresponding ones of the previous proof systems.

**Example**

To illustrate the use of the proof system $\mathcal{R}^R$ we shall now consider the following recursive version of the factorial program:

```plaintext
PROC fac(VAL x)
IS IF x=0 THEN res:=1 ELSE (CALL fac(x-1); res:=x*res)
IN CALL fac(n)
```

Using the data type of Extended Peano Arithmetic and its uniform computational model (Example 3.1-3) we shall construct a proof of the formula

\[
\forall z : \text{TRUE}(\text{factorial: time=11' n+7}) \land \text{TRUE}/(\{n, res\}, 0, 0)
\]

in $\mathcal{R}^R$. We shall present the proof in a combined top-down and bottom-up manner.

We shall first consider the proof for the main program. From /PROC-R/ we get that in order to prove (1) it is sufficient to prove the formula

\[
\forall z : \text{TRUE}(\text{CALL fac(n): time=11' n+7}) \land \text{TRUE}/(\{n, res\}, \text{env}, 0)
\]

where env is the environment defining the procedure fac. The invariant $P(z)$ for the procedure call is now chosen to be the formula $0 \land z = a + 1$ where $a$ is the variable that will be used to hold the value of the call-by-value parameter - confer the rules for the procedure call in the proof system $\mathcal{R}^R$. We then have $\exists z. P(z) \forall a$ so using /cons-R/
we get that it is sufficient to construct a proof of the formula
\[ \exists z. P(z) \land \text{CALL fac(n):time=11\cdot n+7} \rightarrow \text{TRUE}/(\{n, res\}, env, 0). \]

A proof of this formula can be obtained by applying the rule /elim-R/ to a proof of
\[ \forall z. \exists n. \text{CALL fac(n):time=11\cdot n+7} \rightarrow \text{TRUE}/(\{n, res, z\}, env, \{\{fac, z\}\}). \]

Since the time formula \( \text{time}=11\cdot n+7 \) holds if and only if the formula \( (\text{time}=2) \otimes (\text{time}=11\cdot a+5)^n \) holds and since furthermore \( E^S(n) \) is equivalent to \( \text{time}=2 \) and \( \neg P(0) \) holds we get from /CALL-R/ that it is sufficient to prove the formula
\[ P(z) \langle \text{CALL fac(a):time=2} \otimes (\text{time}=11\cdot a+5) \rangle \rightarrow \text{TRUE} \]

(2) \[ P(z+1) \langle \text{IF a=0 THEN res:=1 ELSE (CALL fac(a-1);res:=a\cdot res): time=11\cdot a+5} \rightarrow \text{TRUE}/(\{n, res, z, a\}, env, \{\{fac, z\}\}). \]

The proof of this result for the procedure body will now be presented in a bottom-up manner. In the following let \( \phi \) be an abbreviation for the formula \( P(z) \langle \text{CALL fac(a):time=2} \otimes (\text{time}=11\cdot a+5) \rangle \rightarrow \text{TRUE} \). For the true branch of the conditional it is straightforward to prove that
\[ \psi \rightarrow P(z+1) \land a=0 \langle \text{res:=1:time=11\cdot a+2} \rightarrow \text{TRUE}/(\{n, res, z, a\}, env, \{\{fac, z\}\}) \]

using /ass-R/, /inv-R/ and /cons-R/ (and that we for \( E^S(1) \) can choose \( \text{time}=2 \)). For the false branch we shall first apply the axiom /sel-R/ and get a proof of the formula
\[ \psi \rightarrow P(z) \langle \text{CALL fac(a):time=2} \otimes (\text{time}=11\cdot a+5) \rangle \rightarrow \text{TRUE}/(\{n, res, z, a\}, env, \{\{fac, z\}\}). \]

Using the rule /par-R/ we can then replace the formal parameter \( a \) by \( a-1 \) and we thus get a proof of
\[ \psi \rightarrow P(z) \langle \text{CALL fac(a-1):time=4} \otimes (\text{time}=11\cdot a+5)^{a-1} \rangle \rightarrow \text{TRUE}/(\{n, res, z, a\}, env, \{\{fac, z\}\}). \]

214
since $E^S(a-1)$ is equivalent to $\text{time}=4$. It is straightforward to prove that

$$\phi \Rightarrow \text{TRUE}(\text{res}:=a', \text{res}:\text{time}=4) \Rightarrow \text{TRUE}/(\{n, \text{res}, z, a\}, \text{env}, \{(\text{fac}, z)\})$$

since $E^S(a', \text{res})$ is equivalent to $\text{time}=4$. So using $\text{IF}^R$ and $\text{cons}^R$ with $P(z+1) \land a=0 \Rightarrow P(z)^{a-1}$ and

$$\phi(\text{time}=4) \Rightarrow (\text{time}=11'a+5)^{a-1} \Rightarrow \text{TRUE}'(\text{time}=4) \Rightarrow \text{time}=11'a+2$$

we get a proof of

(4) $$\phi \Rightarrow P(z+1) \land a=0 \Rightarrow \text{CALL fac(a-1); res}:=a' \Rightarrow \text{time}=11'a+2 \Rightarrow \text{TRUE}$$

Applying the rule $\text{IF}^R$ to (3) and (4) gives us the required proof of (2) since $E(a=0)$ can be chosen to be the formula $\text{time}=3$. This completes the proof of (1).

5.3 THE SOUNDNESS THEOREM FOR $\text{R}^R$

As mentioned in the beginning of this chapter the idea is to use lists of assumptions to formalise Hoare’s proofs from assumptions. We are mainly interested in formulas that can be proved without assumptions about other programs, that is, in formulas of the form

$$\Lambda \Rightarrow P(c:RQ/(V, env, L)).$$

When the assumption list is empty the association of procedure names with program variables in $L$ seems unnecessary so we shall mainly be concerned with formulas of the form

$$\Lambda \Rightarrow P(c:RQ/(V, env, \emptyset)).$$

For these formulas we obtain soundness and completeness results similar to those of the previous chapters. We shall in this section consider the soundness property; that of completeness will be handled in the next section.
We have the following result:

**The Soundness Theorem for \( R^R \)**

Given a data type and a numerical computational model for it, if the time expressiveness condition is fulfilled then for every well-formed formula \( \forall \phi \forall c:R \supset Q/(V, \text{env}, \emptyset) \) of \( R^R \)

\[
R^R \forall \phi \forall c:R \supset Q/(V, \text{env}, \emptyset) \text{ implies } \forall \phi \forall c:R \supset Q/(V, \text{env}, \emptyset). 
\]

We shall prove the more general result stating that if the formula \( \phi \supset P<c:R \supset Q/(V, \text{env}, L) \) is a well-formed formula of \( R^R \) then

\[
(\forall \phi) R^R \phi \supset P<c:R \supset Q/(V, \text{env}, L) \text{ implies } \forall \phi \supset P<c:R \supset Q/(V, \text{env}, L). 
\]

In the proof of this result we shall need a few lemmas in addition to those mentioned in Section 5.1.

**Properties of well-formed formulas**

Below we shall prove four lemmas expressing properties of formulas of the form \( P<c:R \supset Q/(V, \text{env}) \) that are well-formed relative to some set \( L \) of procedure name/variable associations. The first two lemmas allow one to extend the \((V, \text{env})\) part of a \( d:L \)-valid formula to contain further variables and further procedure declarations and this in such a way that the validity property is preserved.

**Lemma 5.3-1:** If \( d:L \models P<c:R \supset Q/(V, \text{env}) \) and \( V' \cap V = \emptyset \) then

\[
d:L \models P<c:R \supset Q/(V \cup V', \text{env}). 
\]

**Lemma 5.3-2:** If \( d:L \models P<c:R \supset Q/(V, \text{env}) \), \( q \notin \text{DOM}(\text{env}) \), \( FV(c') \subseteq \{ x \} \cup (V \setminus \text{VAR}(L)) \) and \( FP(c') \subseteq q \cup \text{DOM}(\text{env}) \) then \( d(q=n):L \models P<c:R \supset Q/(V, \text{env}(q=(x, c'))) \) for every natural number \( n \).

The proofs of these two lemmas are straightforward using the lemmas.
The next lemma expresses that if some formula is valid for a maximal recursion depth of \( n+1 \) for a procedure \( p \) then it is also valid when the maximal recursion depth for \( p \) is \( n \):

**Lemma 5.3-3:** If \( d \vdash P \langle c; R \rangle Q/(V, env) \) and \( d(p) = n+1 \) then 
\[ d(p=n) \vdash P \langle c; R \rangle Q/(V, env). \]

The proof of this result is straightforward from the definitions, so we omit the details. Finally, we have the following result:

**Lemma 5.3-4:** If \( d \vdash P \langle c; R \rangle Q/(V, env) \), \( p \in \text{DOM}(env) \rightarrow \text{PROC}(L) \), and \( z \notin V \) then 
\[ d(p=n) \vdash \bigcup \{ (p, z) \mid P \langle c; R \rangle Q/(V \cup \{z\}, env) \} \text{ holds for every natural number } n \text{ provided that the formula } P \langle c; R \rangle Q/(V, env) \text{PROC}(L) \text{ is well-formed relative to } L. \]

Intuitively, this lemma says that if \( p \) is never called when executing \( c \) in the environment \( env \) then we can make any restriction on the depth of the recursive calls of \( p \).

**Proof of Lemma 5.3-4:** In order to prove the lemma we shall first establish two minor properties. First we have

\[ \left\{ \begin{array}{ll} \text{if } P \langle c; R \rangle Q/(V, env) \text{PROC}(L) \text{ is well-formed relative to } L, & (\varepsilon) \\
& \text{then } FP^w(c, env) \subseteq \text{PROC}(L). \end{array} \right. \]

To see this we first observe that the well-formedness condition ensures that \( FP(c) \subseteq \text{PROC}(L) \) and \( FP(env) \subseteq \text{PROC}(L) \). It is then straightforward to prove that \( FP_i(c, env) \subseteq \text{PROC}(L) \) for every \( i \geq 0 \) and thus that \( (\varepsilon) \) holds (remember \( FP(c, env) = \bigcup_{i \geq 0} FP_i(c, env) \)).

Secondly, we have the following result
The proof of this result is by induction on the length of the proof of \((V, env) \vdash (c, s) \rightarrow_d^* s'\) in \(G^R\). The proof is straightforward so we omit the details.

In order to prove Lemma 5.3-4 we must show that

\[(1) \quad d(p=n) : Lu(p,z) \triangleright_p c : R \triangleright Q / (V \cup \exists z)^r, env)\]

holds so assume that \(d(p=n) \triangleright_p c : R \triangleright Q / (V \cup \exists z)^r, env)\) hold for some state \(s\). Then \(s \triangleright_d \exists \) since \(p \triangleright PROC(L)\) and from the assumption of the lemma we get that for some state \(s'\), natural number \(r\) and counter \(d'\)

\[(V, env) \vdash (c, s) \rightarrow_d^* s', \triangleright_p Q(s, s'), \triangleright_R(s, r)\) and \(s \triangleright_d^* d'.\]

Lemma 5.1-3 then gives

\[(V \cup \exists z, env) \vdash (c, s) \rightarrow_d^* s'\]

for some \(s'\) where \(s' \triangleright \exists \). From \(FV(Q) \subseteq \exists \) and \(Q(s, s')\) we get \(Q(s, s')\). We have to prove that \(s \triangleright_d^* \exists \) holds in order to complete the proof of (1). Since \(p \triangleright PROC(L)\) is well-formed relative to \(L\) we get from (8) that \(p \triangleright FP^*(c, env)\) and (8) then gives \(d'(p) = 0\). So from \(s \triangleright_d^* \exists \) we can conclude that \(s \triangleright_d^* \exists \) as required. This completes the proof of Lemma 5.3-3.

**PROOF OF THE SOUNDNESS RESULT**

We now turn to the proof for The Soundness Theorem for \(R^R\), more precisely, the proof of the property (8). We shall prove that the axioms are valid and that the rules preserve validity. The proof showing that the axiom \(/ass-R^R/\) is valid is a straightforward extension of that showing that \(/ass-R/\) is valid in Section 3.3 so we omit...
the details. Similarly, the proofs showing that the rules /IF-\mathcal{R}^R/, /cons-\mathcal{R}^R/ and /inv-\mathcal{R}^R/ preserve validity are straightforward extensions of those showing that /IF-\mathcal{I}/, /cons-\mathcal{I}/ and /inv-\mathcal{I}/, respectively, preserve validity (see Section 3.3). Also here we omit the details. Some extra complications arise for the rules /;\mathcal{R}^R/, /WHILE-\mathcal{R}^R/, /LET-\mathcal{R}^R/ and /PROC-\mathcal{R}^R/ compared with the proofs for the corresponding rules of the proof system \mathcal{R}^N given in Chapter 4 so we shall sketch the proofs below together with those for the new axioms and rules.

Case /;\mathcal{R}^R/: We have to prove that the rule preserves validity so assume that

1. \(P \rightarrow \mathcal{J}^R < c_1 : R_1 > P' \land Q_1 / (V, env, L)\)

and

2. \(P \rightarrow \mathcal{J}^R < c_2 : R_2 > Q_2 / (V, env, L)\).

To prove that

\(P \rightarrow \mathcal{J}^R < c_1 ; c_2 : R_1 \Phi (Q_1 \lor R_2) \lor Q_1 \lor Q_2 / (V, env, L)\)

assume that we for some depth counter \(d\) have

\(d \vdash \mathcal{J}^R / (V, env)\)

for every \(\Phi\) of \(\mathcal{J}^R\). Furthermore, assume that \(s \in L \Gamma d\) and \(P(s)\) holds for some state \(s\). From (1) we then get for some \(s', r\) and \(d'\)

\((V, env) \vdash < c_1, s > \Rightarrow_d s', \mathcal{J}^R \land Q_1 (s, s'), \mathcal{J} R_1 (s, r)\) and \(s \geq L d'\).

Since \(VAR(L) \cap (FV(c) \cup FV(env)) = \emptyset\) we get from Lemma 5.1-1 that \(s \in \text{VAR}(L) s'\) and thereby \(s \in L \Gamma d\). Since \(P'(s')\) holds we can apply assumption (2) and get that for some \(s'', r'\) and \(d''\)

\((V, env) \vdash < c_2, s' > \Rightarrow_d s'', \mathcal{J}^R_2 (s', s''), \mathcal{J} R_2 (s', r')\) and \(s \in L \Gamma d''\).

The semantic rule /;\mathcal{R}^R/ then gives
where \( d_0(p) = \max\{d'(p), d''(p)\} \) for \( p \in \text{DOM}(\text{env}) \). Since \( s \geq_L d', s' \geq_L d'' \) and \( s' \in \text{VAR}(L) \) \( s' \) we get \( s \geq_L d_0 \). It is straightforward to prove that 
\( FQ_0 Q_2(s, s'') \) and \( FR_1 R_2(s, r + r') \). This proves the result. 

Case \( /WHILE-\mathbb{R}/ \): We have to prove that the rule preserves validity 

so assume that 

(1) \( FQ \Rightarrow \forall z. P(z) \langle \text{WHILE } b \text{ DO } c \rangle Q '(V \cup \{z\}, \text{env}, L) \)

and furthermore, that \( FP(0) \Rightarrow \forall b, FP(z) \Rightarrow \forall b, Q \Rightarrow Q, \)
\( FP(z) \Rightarrow b \Rightarrow b \Rightarrow R \), and \( EQ(b) \Rightarrow (Q' \Rightarrow R) \Rightarrow R \) where \( z \) is a variable of sort \( \text{nat} \) satisfying \( z \notin V \). We have to prove that

\( FQ \Rightarrow \exists z. P(z) \langle \text{WHILE } b \text{ DO } c \rangle Q '(V, \text{env}, L) \)

so assume that for some depth counter \( d \)

(2) \( d : L \vdash (V \cup \{z\}, \text{env}) \)

holds for every \( \phi \) of \( \Phi \). By induction on the natural number \( n \) we shall prove that if \( s \leq_L d \) and \( FP(z)(s^n_z) \) hold for some state \( s \) then for some \( s', r \) and \( d' \)

(\$) \( (V, \text{env}) \vdash (\text{WHILE } b \text{ DO } c, s) \overset{d}{\longrightarrow} (s', \forall Q(s, s'), \forall R(s, r) \text{ and } s \geq_L d') \)

Only the induction step is interesting so assume that (\$) holds for \( n = n' \) and that \( FP(z)(s^{n'+1}_z) \) and \( Fb(s) \) hold. Since \( z \notin V(b) (\forall V) \) we have \( FP(z+1) \Rightarrow b(s^{n'}_z) \) and furthermore \( s^{n'}_z \leq_L d \) since \( z \notin \text{VAR}(L) \) and \( s \leq_L d \).

From (2) and Lemma 5.3-1 we get

\( d : L \vdash (V \cup \{z\}, \text{env}) \)

for every \( \phi \) of \( \Phi \) and we can apply (1) and get that for some \( s', r_1 \) and \( d_1 \)

\( (V \cup \{z\}, \text{env}) \vdash (c, s^n_z) \overset{d_1}{\longrightarrow} (s', FP(z)Q'(s^n_z, s'), FR'(s^n_z, r_1) \text{ and } s^{n'}_z \geq_L d_1) \).
Since $(\{z\} \cup \text{VAR}(L)) \cap (\text{FV}(c) \cup \text{FV}(\text{env})) = \emptyset$, we can apply Lemma 5.1-1 and get $s'^n \vdash_{L} s'$ and thereby $s' \vdash_{L} d$ and $\vdash P(z)(s'^n)$. We can then apply the induction hypothesis. The rest of the proof is now a straightforward modification of the proof in Section 4.3 showing that the rule \(\text{WHILE-}\mathcal{N}\) preserves validity so we omit the details.

Case /LET-N\(^R\)/: We shall prove that the rule preserves validity so assume that

\[
\begin{align*}
(1) & \quad \vdash P(\forall x' = e\langle c_{x'}^x : R\rangle Q / (V\cup x', \text{env}, L))
\end{align*}
\]

where $x'$ is a variable of the same sort as $x$ satisfying that $x' \notin \text{VFV}(Q)$. To prove

\[
\begin{align*}
\vdash P(\forall x' = e\langle c_{x'}^x : R\rangle Q / (V\cup x', \text{env}, L))
\end{align*}
\]

assume that for some depth counter $d$

\[
\begin{align*}
d : L \vdash \phi / (V, \text{env})
\end{align*}
\]

holds for every $\phi$ of $\overline{\phi}$. From Lemma 5.3-1 we then get

\[
\begin{align*}
d : L \vdash \phi / (V\cup x', \text{env})
\end{align*}
\]

and we can now apply the assumption (1) and get

\[
\begin{align*}
d : L \vdash P(\forall x' = e\langle c_{x'}^x : R\rangle Q / (V\cup x', \text{env})).
\end{align*}
\]

The proof showing that

\[
\begin{align*}
d : L \vdash P(\forall x' = e\langle c_{x'}^x : R\rangle Q / (V, \text{env})
\end{align*}
\]

holds is a straightforward modification of that proving that /LET-N\(^R\)/ preserves validity (see Section 4.3) so we omit the details.

Case /PROC-N\(^R\)/: We have to prove that the rule preserves validity so assume that

\[
\begin{align*}
(1) & \quad \vdash P(c_{2p}^q : R\rangle Q / (V, \text{env}(q=(x,c_{1p}^q)), L))
\end{align*}
\]

221
where q is a procedure name satisfying q ∈ DOM(env). To prove

$$\text{PROC } p(\text{VAL } x) \text{ IS } c_1 \text{ IN } c_2:R \Rightarrow Q/(V, env, L)$$

assume that for some counter d

(2) \(d: L \vdash \phi/(V, env)\)

holds for every \(\phi\) of \(L\) and that \(s \models L\) and \(\models P(s)\) hold for some state \(s\).

Since \(q \in \text{DOM}(env)\), \(FV(c_1^q) \subseteq \text{V-VAR}(L)\) and \(FP(c_1^q) \subseteq q \in \text{DOM}(env)\) we can apply Lemma 5.3-2 and get from (2) that

$$d(q=0): L \vdash \phi/(V, env(q=(x, c_1^q))).$$

Since \(q \in \text{PROC}(L)\) (\(q \in \text{DOM}(env)\)) we get from \(s \models L\) that \(s \models L(q=0)\). Using the assumption (1) we therefore get

$$(V, env(q=(x, c_1^q))) \vdash \langle c_2^q, s \rangle \vdash \langle s', Q(s, s'), R(s, r) \rangle$$

for some \(s', r\) and \(d'\). The semantic rule /PROC/ then gives

$$(V, env) \vdash \text{PROC } p(\text{VAL } x) \text{ IS } c_1 \text{ IN } c_2:R \Rightarrow S'$$

where \(d''(p')=d'(p')\) for \(p' \in \text{DOM}(env)\). Clearly, we have \(s \models L\) and since both \(Q(s, s')\) and \(R(s, r)\) hold we have proved the required result.

Case /CALL-\(\phi\): We have to prove that the rule preserves validity so assume that

(1) \(L \vdash \text{CALL } p(a):E^a(a) \Rightarrow Q \Rightarrow P(z+1)\langle c_x^a, x \rangle \Rightarrow Q/(V \cup \{a\}, env, L)\)

and

(2) \(L \vdash \phi\)

where \(env(p)=(x, c)\), \((p, z) \in L\), \(a\) is a variable of the same sort as \(x\) satisfying \(a \notin \text{V-FV}(Q)\) and \(z\) is a variable of sort \text{nat} satisfying \(z \notin \text{FV}(R)\) and \(z \notin \text{FV}(Q)\). We shall prove that
(3) \[ P(z)^a \langle \text{CALL } \text{p}(a):E^S(a)\triangleright Q^a \rangle / (V,env,L). \]

By induction on the natural number \( n \) we shall prove that if \( d(p) = n \) then

\[
\begin{cases}
\text{if } d(L\phi/(V,env) \text{ holds for every } \phi \text{ of } \phi \text{ then} \\
 d:LP(z)<\text{CALL p(a):E}^S(a)\triangleright Q/(V\cup\{a\},env).
\end{cases}
\]

First we assume that \( n = 0 \) and that \( sLd \) and \( \triangleright P(z)(s) \) hold for some state \( s \). Then \( z(s) = 0 \) must hold and thereby \( \triangleright P(0)(s) \). This is a contradiction with the assumption (2) so \((\$)\) holds vacuously if \( d(p) = 0 \).

For the induction step assume that \((\$)\) holds for \( n = n' \) and we shall prove it for \( n = n' + 1 \). Assume now that \( d:LP(z)<\text{CALL p(a):E}^S(a)\triangleright Q/(V\cup\{a\},env) \)
holds for every \( \phi \) of \( \phi \) and that \( d(p) = n' + 1 \). From Lemma 5.3-3 we then get

\[ d(p=n') : L\phi/(V,env) \]

and the induction hypothesis gives that

\[ d(p=n') : LP(z+1)<\text{CALL p(a):E}^S(a)\triangleright Q/(V\cup\{a\},env). \]

From the assumption (1) we then get

\[ d(p=n') : L\phi/(V,env) \]

To prove

\[ d:LP(z)<\text{CALL p(a):E}^S(a)\triangleright Q/(V\cup\{a\},env) \]

assume that \( sLd \) and \( \triangleright P(z)(s) \) hold for some state \( s \). Then \( z(s) \neq n' + 1 \) and since \( z(s) = 0 \) cannot occur (contradicts the assumption (2)) we have \( z(s) = n'' + 1 \) for some natural number \( n'' \). We have \( \triangleright P(z+1)(s_{z}^{n''}) \) and since \( n'' \neq n' \) we also have \( s_{z}^{n''} \neq d(p=n'). \) Therefore we get from (5) that for some \( s', r \) and \( d' \)

\[ (V\cup\{a\},env)\vdash a_{x} \langle x_{s}^{n''} \triangleright d, s', \triangleright Q(s_{z}^{n''},s'), \triangleright R(s_{z}^{n''},s') \rangle / (V\cup\{a\},env). \]
Let now $a'$ be a new variable of the same sort as $x$ and $a$, that is, $a' \notin \text{FV}(a)$. Define $s_0$ to be the state $s_a(s)$. Then $s_{0a} = \text{FV}(a)s_n$ and since $z \notin \text{FV}(c^a_x) \subset \text{FV}(\text{env})$ (follows from the well-formedness of the formula (1)) we get from Lemma 5.1-2 that

$$(\text{FV}(a), \text{env}) \vdash \langle c^a_x, s_0a \rangle \rightarrow_d s'_2$$

where $s'_2 \leq s'_1$. Because $a' \notin \text{FV}(a)$ we can rename $a$ to $a'$ and get using Lemma 5.1-5 that

$$(\text{FV}(a), a'), \text{env} \vdash \langle c^a_x, s_0a \rangle \rightarrow_d s'_3$$

where $s'_3 \leq s'_2$. Since $a' \notin \text{FV}(\text{env}) (\leq \forall)$ this means that

$$(\text{FV}(a), a'), \text{env} \vdash \langle c^a_x, s(s) \rangle \rightarrow_d s'_3.$$  

The semantic rule /CALL-/ then gives

$$(\text{FV}(a), \text{env}) \vdash \langle \text{CALL} p(a), s \rangle \rightarrow_a^d (s(a) + a(s) + r) \rightarrow_d s'_3$$

where $d'' = d'(p=d'(p)+1)$.

In order to complete the proof of (6) we have to prove that

$\forall (s, s'_3), \forall \text{FV}(a) \Theta R(s, a(s) + a(s) + r)$ and $s \leq d''$. We have $\text{FV}(a) \subseteq (\text{FV}(\text{env}) - \{a\})$ and since $\forall (s_n^m, s'_3)$ and $s'_3 \leq s'_3$ we get $\forall (s, s'_3)$. We have $\forall \text{FV}(\text{env}) \subseteq (V - \{z\} \cup \text{time})$ and since $\forall R(s_n^m, r)$ holds we get $\forall R(s, r)$. The time expressiveness assumption together with Lemma 3.2-1 gives that $\forall \text{FV}(a)(s, a(s) + a(s) + r)$ holds and thereby $\forall \text{FV}(a) \Theta R(s, a(s) + a(s) + r)$. We have $s_n^m \leq d$ and since $d''(p) = d'(p) + 1 \leq n'' + 1 = z(s)$ we get $s \leq d''$. This proves (6) and thereby (9).

Finally, we shall prove that (3) holds. So assume that we for some counter $d$ have

(7) $d \leq \bar{\phi}/(V, \text{env})$

for every $\phi$ of $\bar{\phi}$. Furthermore assume $s \leq d$ and $\forall P(z) \subseteq (s)$ hold for some
state s. Then $P(z)(s_a^o(s))$ and since $a \in \mathcal{V}$ we have $s_a^e(s) \not\leq_L d$. From (5) and (7) we therefore get that for some $s', r$ and $d'$

$$(V, \mathcal{a}, \mathcal{v}) \vdash \langle \text{CALL } p(a), s_a^e(s) \rangle \gamma \sum_{a} d', s', \bullet Q(s_a^e(s), s'),$$

$$(V, \mathcal{a}) \emptyset R(s_a^e(s), r) \text{ and } s_a^e(s) \not\leq_L d'.$$

Now we can apply Lemma 5.1-7 and get that $r = a^+(s_a^e(s) + a(s_a^e(s)) + r')$ for some $r'$ and

$$(V, \mathcal{a}, \mathcal{v}) \vdash \langle \text{CALL } p(e), s \rangle e^+(s) + e(s) + r', s',$$

where $s' \nless s'$. We can then apply Lemma 5.1-2 and get

$$(V, \mathcal{v}) \vdash \langle \text{CALL } p(e), s \rangle e^+(s) + e(s) + r', d_2, s'_1$$

where $s'_1 \nless s'$ and since $Q(s_a^e(s), s')$, $F V(Q_a^e(s), s') \nless v \nless s'$ and $s'_1 \nless s'$ we get

$$(V, \mathcal{v}) \vdash \langle \text{CALL } p(e), s \rangle e^+(s) + e(s) + r', d_2, s'_1$$

where $s'_1 \nless s'$ and since $Q(s_a^e(s), s')$, $F V(Q_a^e(s), s') \nless v \nless s'$ and $s'_1 \nless s'$ we get

$$(V, \mathcal{v}) \vdash \langle \text{CALL } p(e), s \rangle e^+(s) + e(s) + r', d_2, s'_1$$

where $s'_1 \nless s'$. From $r = a^+(s_a^e(s) + a(s_a^e(s)) + r')$ and $E^+(a) \emptyset R(s_a^e(s), r)$ we get $E^+(a) \emptyset R(s_a^e(s), r')$ using the time expressiveness assumption and Lemma 3.2-1.

So we have $E^+(a) \emptyset R(s_a^e(s), r')$. From the time expressiveness assumption and Lemma 3.2-1 we also have $E^+(e) \emptyset R(s_a^e(s), e^+(s) + e(s) + r')$ and thereby

$$(V, \mathcal{v}) \vdash \langle \text{CALL } p(e), s \rangle e^+(s) + e(s) + r', d_2, s'_1$$

where $s'_1 \nless s'$. This proves (3) and thereby that the rule 

$$(\text{CALL}_a^R)$$

preserves validity.

Case /sel-\[X^R]: It is straightforward to prove that the axiom is valid, that is, that

$$\not\exists P(\langle c : R \rangle Q / (V, \mathcal{v}))$$

where $P(\langle c : R \rangle Q)$ is in $G_1$. We omit the details.

Case /par-\[X^R]: We have to prove that the rule preserves validity so assume that

(1) $\not\exists P(\langle \text{CALL } p(a) : e^+(a) \emptyset R \rangle Q / (V, \mathcal{v}))$

where $a \in \mathcal{V} \cup (F V(\mathcal{v})) \cup F V(Q)$. To prove
assume that for some counter d

(2) \( d : L \psi / (V, \text{env}) \)

holds for every \( \phi \) of \( \Phi \). Furthermore, assume that \( s \not\in d \) and \( \exists P_a^e(s) \) hold for some state \( s \). Then \( \exists P_a^e(s) \) holds and since \( a \not\in \text{VAR}(L) \) (follows from the well-formedness of (1)) we have \( s_a^e(s) \not\in d \). So from (1) we get that for some \( s', r \) and \( d' \)

\[
(V, \text{env}) \models \langle \text{CALL } p(e), s_a^e(s) \rangle \xrightarrow{d \vdash r} s', \langle \text{CALL } e(s), s' \rangle, \text{ and } s_a^e(s) \not\in d'.
\]

From Lemma 5.1-7 we get that \( r = a (s_a^e(s) + a(s_a^e(s) + r') \) for some \( r' \) and

\[
(V, \text{env}) \models \langle \text{CALL } p(e), s_a^e(s) + e(s) + r' \rangle \xrightarrow{d \vdash r} s'.
\]

where \( s = \{ a \} \rightarrow s' \). From \( \exists Q_a^e(s) \) and \( \text{FV}(Q_a^e) \subseteq \{ v \} \) we get \( \exists Q_a^e(s, s') \). From \( \exists E_a^e(s) \) we get \( r = a (s_a^e(s) + a(s_a^e(s) + r') \) and thereby we get \( \exists R_a^e(s, s_a^e(s) + e(s) + r') \). We have \( s_a^e(s) \not\in d' \) and since \( a \not\in \text{VAR}(L) \) this implies that \( s \not\in d' \). This proves the required validity.

\[
\text{Case } /\text{elim-R}/: \text{ We shall prove that the rule preserves validity so assume that}
\]

(1) \( \exists P_a^e \psi \langle c : R \rangle \Theta (V u \{ z_1, \ldots, z_k \}, \text{env}, L u L' ) \)

where \( L u L' = \emptyset \), \( \text{VAR}(L') \not\subseteq \emptyset \) and \( \text{VAR}(L) = \{ z_1, \ldots, z_k \} \). We shall prove that

(2) \( \exists P_a^e \exists z_1 \ldots \exists z_k . \psi \langle c : R \rangle \Theta (V, \text{env}, L ) \)

so assume that for some counter d
(3) \(d: L \phi / (V, \text{env})\)

holds for every \(\phi\) of \(\hat{\Phi}\). Furthermore, assume that \(s \in L d\) and

\(\vdash (\exists z_1 \ldots \exists z_k : P)(s)\) hold for some state \(s\). Then there are natural numbers \(n_1 \ldots n_k\) such that \(\forall P(s_0)\) holds for \(s_0\) being \(z_1 \ldots z_k\).

Define \(d_0\) to be the counter with

\[
d_0(p) = \begin{cases} 
n_i & \text{if } (p, z_i) \in L' \text{ for some } i, 1 \leq i \leq k \\
(d(p) & \text{otherwise.}
\end{cases}
\]

Since \(L \cap L' = \emptyset\) and \(s \notin d\) we have \(s_0 \notin L d_0\). From Lemma 5.3-4 and (3) we get

\[
d_0 : LuL' \vdash \phi / (V \cup \text{VAR}(L'), \text{env})
\]

holds for every \(\phi\) of \(\hat{\Phi}\). Then the assumption (1) gives that for some

\[
s' , r\text{ and } d'
\]

\[(V \cup \text{VAR}(L'), \text{env}) \vdash \langle c , s_0 \rangle \frac{L_d}{d} , s' , \vdash Q(s_0, s') , \vdash R(s_0, r)\text{ and } s_0 \geq LuL' d' .
\]

Using Lemma 5.1-2 with \(s_0 \xrightarrow{L} s\) we get

\[(V, \text{env}) \vdash \langle c, s \rangle \frac{L_d}{d} , s''
\]

for some \(s''\) with \(s'' \approx V s'\). Since \(\text{FV}(Q) \subseteq V \cup \text{V} \) and \(\vdash Q(s_0, s')\) we get \(\vdash Q(s, s'')\).

Since \(\text{FV}(R) \subseteq V \cup \{\text{time}\}\) and \(\vdash R(s_0, r)\) we get \(\vdash R(s, r)\). Finally, from

\(s_0 \geq LuL' d'\) we get \(s \geq d'\). This completes the proof of (2) and thereby that the rule \(\text{/elim-R}/\) preserves validity.

Case \(\text{/ext-R}/\): To prove that the rule preserves validity assume that

(1) \(\vdash \hat{\Phi} \Rightarrow P \langle \text{CALL } p(e) : R \rangle Q / (V, \text{env}, L)\)

and that for some counter \(d\)

\(d: L \phi / (V, \text{env})\)

for every \(\phi\) of \(\hat{\Phi}\). We shall prove that

\(d: L \phi \langle \text{CALL } p(e) : R \rangle Q I_{V, \text{env}} / (V, \text{env})\)
for \( V' \subseteq V - FV(\text{env} \cup \text{PROC}(L)) \). Furthermore assume that \( s \in L^d \) and that \( \forall p(s) \) hold for some state \( s \). From (1) we get

\[(V, \text{env}) \vdash \langle \text{CALL } p(e), s \rangle \xrightarrow{d'} s', \forall q(s, s'), \forall r(s, r) \text{ and } s \in L^d' \]

for some \( s', r \) and \( d' \). From the semantic rule /CALL-\( g^R \) we get that

\[ r = e^S(s) + e^S(s) + r' \text{ for some } r' \] and

\[ (V \cup \{x', \}, \text{env}) \vdash \langle c^{x'}, x', s_x^e(s), r', d'' \rangle \rightarrow d'' \]

where \( \text{env}(p) = (x, c) \), \( x' \) is a variable of the same sort as \( x \) satisfying \( x' \notin V \) and \( d'' = d'(p = d'(p) - 1) \). Since \( V' \cap FV(c^{x'}) \cup FV(\text{env} \cup \text{PROC}(L)) = \emptyset \) follows from \( V' \cap FV(\text{env} \cup \text{PROC}(L)) = \emptyset \) and \( V' \subseteq V \) we can apply Lemma 5.1-1 and get \( s_x^e(s) \subseteq V, s' \) and thereby \( \forall I_V, (s, s') \). This proves the required result.

This completes the proof of The Soundness Theorem for \( R^R \).

### 5.4 THE COMPLETENESS THEOREM FOR \( R^R \)

We now turn to the proof of a completeness result for the proof system \( R^R \) for analysing the run-time of recursive procedure programs. The expressiveness concept used when proving a completeness result for \( R^R \) is essentially as that used in the completeness result for \( N \), see Section 4.4. So given a data type and a numerical computational model for it we say that the expressiveness condition for \( R^R \) is fulfilled if the time expressiveness condition is and furthermore, for every procedure program \( c \), every finite set \( V \) of program variables and every environment \( \text{env} \) with \( FV(c) \cup FV(\text{env}) \subseteq V \) and \( FP(c) \cup F\text{FP}(\text{env}) \subseteq \text{DOM}(\text{env}) \)

- there exists a relational formula \( G \in V \cup \text{env} \) with \( FV(G) \cup \{c\} \subseteq \text{V} \)

\( V \cup \text{V} \) satisfying that for each pair \((s, s')\) of states

\[ \forall G \in V \cup \text{env} \cup \{c\}(s, s') \]

228
if and only if

\[(V,\text{env}) \vdash <c,s> \xrightarrow{d} s'' \text{ for some } s'', r \text{ and } d \text{ with } s'' \models s',\]

and

- there exists a time formula \(E_{V,\text{env}}[c] \) with \(\text{FV}(E_{V,\text{env}}[c]) \subseteq \text{time}\) satisfying that for every pair \((s,r)\) of state and natural number

\[\xi_{E_{V,\text{env}}[c]}(s,r)\]

if and only if

\[(V,\text{env}) \vdash <c,s> \xrightarrow{d} s' \text{ for some } s' \text{ and } d.\]

Using this expressiveness concept we shall prove the following result:

**The Completeness Theorem for \(\mathcal{R}^R\)**

Given a data type and a numerical computational model for it, if the expressiveness condition for \(\mathcal{R}^R\) is fulfilled then for every well-formed formula \(\forall c:R \forall (V,\text{env},\emptyset)\) of \(\mathcal{R}^R\):

\[\forall \forall P(c:R)/\forall (V,\text{env},\emptyset) \implies \mathcal{R}^R \vdash \forall P(c:R)/\forall (V,\text{env},\emptyset). ///\]

In the completeness proof for \(\mathcal{R}^N\), given in Section 4.4 we proceed by induction on the possible depth of procedure calls. This approach does not work here because procedures might call each other recursively. However, the assumption lists can be used to hold information procedures that are called recursively and this will be used in the proof below.

**Construction of Assumption Lists**

To get an idea of how the assumption lists can be used in a proof in the proof system \(\mathcal{R}^R\) and thereby in the proof of the completeness result we shall consider a program of the following form:
Initially, we assume that the environment is empty and furthermore that the assumption list is empty.

Let us first consider the three non-recursive calls of the procedure $p$, that is, the calls at the program points $C$, $D$ and $E$ above. At the program point $C$ the environment will only contain the declaration of $p$. Using the rule /CALL-$X^R$/ we will add a formula, say $\phi_p(z_p)$, to the assumption list and we will start the analysis of the body of $p$; here $z_p$ is the variable that counts the recursion depth for the procedure $p$. When the analysis of the body of $p$ is completed the rule /CALL-$X^R$/ ensures that $\phi_p(z_p)$ is removed from the assumption list.

At the program point $E$ we have extended the environment so it contains both the declaration of $p$ and $r$. Using the rule /CALL-$X^R$/ we
we will again add the formula $\phi_p(z_p)$ to the assumption list and it will be removed when the analysis of the body of $p$ has been completed.

At the program point $E$ we call the procedure $r$. Since $r$ may call $p$ (at the point $D$) we want to bound the recursion depth for any procedure of the environment that might be called. We shall therefore add a formula, say $\phi_r(z_p,z_r)$, to the assumption list. It expresses a property of a call of $r$ but takes into account both the recursion depth $z_p$ of $p$ and that of $r$, $z_r$.

When analysing the body of the procedure $r$ we thus have the assumption list $\phi_r(z_p,z_r)$. At the program point $D$ we call the procedure $p$ and we shall here add the formula $\phi_p(z_p)$ to the assumption list. The body of the procedure $p$ is then analysed with the assumption list $\phi_r(z_p,z_r)\&\phi_p(z_p)$.

At the program point $B$ we have an environment that contains the declaration of all the three procedures $p$, $q$ and $r$ (although we are not within $r$'s scope). At the point $B$ we shall apply the rule $\text{/CALL-R}$/ and add a formula $\phi_q(z_p,z_q)$ to the assumption list. This formula expresses a property of a call of the procedure $q$ and since $q$ may call $p$ as well as $q$ it will take into account both the recursion depth $z_p$ of $p$ and that of $q$, $z_q$. The procedure $r$ cannot be called so it is ignored.

Finally, at the program point $A$ we call the procedure $p$. The assumption list will now have the form $\phi_r(z_p,z_r)\&\phi_p(z_p)\&\phi_q(z_p,z_q)$ so it contains information about $p$ and we can apply the selection axiom and complete the proof.

This example analysis has illustrated some important properties of
the assumption lists. At any point in the proof the assumption list corresponds very closely to the dynamic link in a stack implementation of recursive procedures (however, with the important difference that recursive calls do not cause extra information to be added to the assumption list; in the implementation a new activation record is pushed on the top of the stack for each call). When we add a formula \( \phi \) expressing a property of the procedure \( p \) to the assumption list we require that \( \phi \) takes into account the recursion depth of all (known) procedures that may be called by \( p \). This is exactly the procedures on the static link in the stack implementation mentioned above.

Consider now a (reasonable) environment \( \text{env} \) and let \( \langle \rangle \) be the relation on \( \text{DOM}(\text{env}) \) defined by

\[
\langle p, q \rangle \text{ if and only if } q \in \text{FP}(\text{c}) \text{ where } \text{env}(p) = (x, c).
\]

That is, \( \langle p, q \rangle \) holds if \( p \) may call \( q \). Any sequence \( p_1, \ldots, p_k \) of procedure names satisfying \( p_i \prec p_{i+1} \) (for \( 1 \leq i < k \)) will represent a possible dynamic link. We shall especially be interested in the lists that do not contain repetitions as they, according to the discussing above, correspond closely to the assumption lists. So we define:

A **trace list** over the environment \( \text{env} \) is a sequence \( p_1 \ldots p_k \) of procedure names from \( \text{DOM}(\text{env}) \) satisfying

- \( p_i \prec p_{i+1} \) for \( 1 \leq i < k \),
- if \( p_i \not\prec p_j \) then \( i = j \) (for \( 1 \leq i, j \leq k \)).

Note that if \( l \) is a trace list over the environment \( \text{env} \) then it will also be a trace list over any extension of \( \text{env} \).

Given a (non-empty) trace list \( l \) we shall construct another list \( I \) of procedure names that corresponds to the static link of \( l \): \( I \) is the list of procedure names obtained from \( l \) by removing all procedure
names q from l that does not satisfy \( q \preceq^* p \) where p is the right-most procedure name occurring in the list l. Remember, \( \preceq \) is the ordering on \( \text{DOM}(\text{env}) \) defined by

\[ q \preceq p \text{ if and only if } \text{env}(p) = (x,c) \text{ and } q \in \mathcal{FP}(c) - \{p\} \]

and \( \preceq^* \) is the transitive reflexive closure of \( \preceq \).

**Example 5.4-1:** Let us for a moment return to the program sketched earlier in this section. During the execution of this program we may reach the program point A with the trace list \( \text{rpq} \) (reflecting that a call of r, a call of p and a call of q are active at that point of time). The environment contains the declaration of the three procedures p, q and r and the relation \( \preceq \) is given by \( p \preceq q \) and \( p \preceq r \). The "static link" corresponding to the trace list \( \text{rpq} \) is therefore \( \text{pq} \).

In the completeness proof we shall construct the assumption lists from the trace lists and the formulas that are added to the assumption list during the proof will also be defined on the basis of the trace lists. Let us explain this further.

Consider an environment \( \text{env} \) and a set \( V \) of variables satisfying \( \text{FV}(\text{env}) \subseteq V \). To each procedure name \( p \) of \( \text{DOM}(\text{env}) \) we associate two distinct variables \( z_p \) and \( a_p \); \( z_p \) has sort \( \text{nat} \) and \( a_p \) has the same sort as the formal parameter of \( p \) in the declaration of \( p \) specified by \( \text{env} \). We shall require that \( z_p, a_p \notin V \). The idea is that \( z_p \) counts the recursion depth for the procedure \( p \) whereas \( a_p \) stands for the actual value of the call-by-value parameter of the procedure in a given call. Given a trace list \( l \) with \( p \) as the right-most procedure name we shall define a formula \( \phi_1 \) of the form

\[
\begin{align*}
\phi_1(\text{rp}) \langle \text{CALL } p(a_p); E^S(a_p)\Theta_R \rangle_{Q_1}
\end{align*}
\]
Intuitively, $P_1(z)$ ensures that the program CALL $p(a_p)$ terminates in such a way that the recursion depth of the procedures named in the list $I$ are bounded by the values of their respective $z$-variables. So especially, $z_p$ bounds the recursion depth of the calls of $p$. To define $P_1(z)$ formally we shall use a trick similar to that used in the completeness proof of Chapter 2 for the while construct. We shall transform the body of each procedure $p'$ of $I$, say $c'$, into a program $c'_p$ that keeps track of the depth of the recursive calls of $p'$ and loops if it becomes too great. We define $c'_p$ to be the program

$$\text{IF } 0(z_p, \text{THEN } (z_p; z_p; 1; c'; z_p; z_p; 1) \text{ ELSE loop}$$

where we define $z_p; z_p; 1$ and loop to be the two programs:

- LET $z=0$ IN (WHILE $z+1(z_p, \text{DO } z:=z+1; z_p:=z$) (where $z\neq z_p$)
- WHILE TRUE DO $z_p:=z_p,$ respectively. Clearly, $c'_p$ is a procedure program.

Given the trace list $l$ we now define a new environment $env_1$ by

$$env_1(p') = \begin{cases} (x', c'_p) \text{ if } p' \text{ is in } I \text{ and } env(p')=(x', c') \\ env(p') \text{ otherwise.} \end{cases}$$

This means that the procedures mentioned in $I$ keep track of their recursion depth in $env_1$. Define now the two sets $L_1$ and $V_1$ by

$$L_1 = \{(p', z_p) | p' \text{ is in } l\}$$

and

$$V_1 = \bigcup \{a_p, [p' \text{ is in } l] \cup \{z_p, [p' \text{ is in } l]\}.$$ 

The formulas $P_1(z)$, $Q_1$ and $R_1$ of $\phi_1$ are now defined as follows

$$P_1(z)^\exists \{x', C_{env_1}^{\{\text{CALL } p(a_p)\}}^{X'} x \} x \frac{x}{x}$$
where $x$ is a vector of the variables of $V_1$, $\overline{x}$ is the corresponding vector of shadow variables and $X'$ is a vector of new distinct variables of appropriate sorts and of the same length as $X$. Furthermore,

$$Q_1 \equiv \exists p. G_{V_1-VAR(L_1),env}^{\text{CALL } p(a_p)}$$

and

$$R_1 \equiv E_{V_1-VAR(L_1),env}^{a_x^p}$$

where $env(p) = (x,c)$. Note that $a^p \in FV(Q_1)$ and if $z \in VAR(L_1)$ then $z, a^p \notin FV(Q_1)$ as well as $z \notin FV(R_1)$.

We shall now prove that for any depth counter $d$ defined on the domain of the environment $env$ we have $d: L_1 \vdash \phi_1/(V_1, env)$. However, in order to do that we shall first prove a few lemmas. First we have

**Lemma 5.4-1:** If $(V_1, env) \vdash (c', s) \not\vdash_d s'$ for some program $c'$ and environment $env$ with $z^p \in FV(c') \cup FV(env)$ and where $z^p(s) \not\vdash_d (p)$ then for some $s''$ and $r'$ $(V_1, env(p = (x, c_p))) \vdash (c', s) \not\vdash_d s''$ and $s'' \equiv_{V_1-\overline{V}_1} z^p(s')$

$z^p(s'') = z^p(s)$ and the two proofs in $\Omega^R$ have the same lengths. //

The proof of this result is sketched in Appendix C. As a corollary we have the following result

**Lemma 5.4-2:** If $(V_1, env) \vdash (c, s) \not\vdash_d s'$ for some program $c$ with $FV(c) \subseteq V_1-\overline{V}_1$ and if $s \not\vdash_d d$ then $(V_1, env_1) \vdash (c, s) \not\vdash_d s''$ for some $s''$ and $r'$.

This result tells us how to replace an environment $env$ by $env_1$. The following lemma will show how to replace $env_1$ by $env$:

**Lemma 5.4-3:** If $(V_1, env_1) \vdash (c, s) \not\vdash_d s'$ for some program with $FV(c) \subseteq V_1-\overline{V}_1$ and $FP(c) \subseteq \text{PROC}(L_1)$ then $s \not\vdash_d d$, $d(p') = 0$ for $p' \notin \text{PROC}(L_1)$ and $(V_1, env) \vdash (c, s) \not\vdash_d s''$ for some $s''$ and $r'$ where $s'' \equiv_{V_1-\overline{V}_1} s'$. //

235
The proof of this result is sketched in Appendix C. It is straightforward to verify that the following lemma holds

**Lemma 5.4-4:** If \( l = l'^q l'' \) then the formula \( \phi_{l',q'}(V_1,env,Proc(L_1)) \) is well-formed relative to \( L_1 \).

Furthermore, we have

**Lemma 5.4-5:** If \( l = l'^q l'' \) then for any counter \( d \), \( d : L_1 \phi_{l',q'}(V_1,env) \).

**Proof:** Assume that \( s \in L_1 \) and \( \text{FP}_{l',q'}(z_q)(s) \) hold for some state \( s \). The definition of \( P_{l',q'}(z_q) \) and the expressiveness assumption then give that for some \( s', d' \) and \( r' \)

\[
(V_1'q',env_1'q') \vdash \langle \text{CALL } q(a), s \rangle \xrightarrow{d'} s'.
\]

From Lemma 5.4-3 we then get that \( s \in L_1'q' \), \( d' = d' \), \( d'(p') = 0 \) for \( p' : Proc(L_1',q) \) and

\[
(V_1'q',env) \vdash \langle \text{CALL } q(a), s \rangle \xrightarrow{d'} s_1'.
\]

for some \( r' \) and \( s_1' \). Since \( l = l'^q l'' \) we have \( V_1'q \subseteq V_1 \) and from Lemma 5.1-3 we get that for some \( s_2' \)

\[
(V_1,env) \vdash \langle \text{CALL } q(a), s \rangle \xrightarrow{d'} s_2'.
\]

where \( s_2' = V_1'q' \).

We have to prove that \( \exists Q_{l',q}(s_1',s_2') \), \( \exists t E(a) \Theta_{L_1'q}(s, r') \) and \( s \in L_1 \) hold and the proof of the lemma will then be completed. First, from

(1) and Lemma 5.1-2 we get that

\[
(V_1'q',\text{VAR}(L_1'q'),env) \vdash \langle \text{CALL } q(a), s \rangle \xrightarrow{d'} s_1'.
\]

where \( s_1' = V_1'q' \). Since \( \text{VAR}(L_1'q) \cap (\{a_q\} \cup \text{FV}(env)) = \emptyset \). The expressiveness assumption together with the definition of \( Q_{l',q} \) gives that

\[ \exists Q_{l',q}(s,s') \) holds. Since \( s_3' = V_1'q' \) \( \text{VAR}(L_1'q)'s_2' \) and \( \text{FV}(Q_{l',q}) = \emptyset \).
From (2) and the semantic rule /CALL-\$R/ we get that for some r" given by 
\[ r' = a^q(s) + a^q(s)^+ + r" \]
we have
\[(V_1^q \text{VAR}(L_1^q)) \cup \{x'\}, \text{env}\} \vdash \langle c^x', s', x', s \rangle \overset{r"}{\rightarrow} d' \rightarrow s_5' \]

where d" = d"(q = d"(q) + 1), env(q) = (x, c) and x' is a variable of the same sort as x satisfying \[ x' \notin V_1^q \text{VAR}(L_1^q). \] Since \[ a^q FV(c^x') \cup FV(env) \]
\[(\Sigma V \cup \{x'\}) \] we can apply Lemma 5.1-2 and get that for some s_4'
\[(V_1^q \text{VAR}(L_1^q)) \cup \{a^q\}) \cup \{x'\}, \text{env}\} \vdash \langle c^x', s', x', s \rangle \overset{r"}{\rightarrow} d' \rightarrow s_4'. \]

Then Lemma 5.1-5 can be used to rename x' to a and we get
\[(V_1^q \text{VAR}(L_1^q)) \cup \{a^q\}) \vdash \langle c^x', s', x', s \rangle \overset{r"}{\rightarrow} d' \rightarrow s_5' \]

for some s_5'. Since \[ x' \notin V \] and \[ FV(env) \notin c \] we get from Lemma 5.1-2 that
\[ (V_1^q \text{VAR}(L_1^q)), \text{env}\} \vdash \langle c^x', s', x', s \rangle \overset{r"}{\rightarrow} d' \rightarrow s_6' \]

for some s_6'. The expressiveness assumption together with the definition of \[ R_1^q \] now gives that \[ \exists R_1^q(s, r''). \] Since the time expressiveness assumption is fulfilled we get from Lemma 3.2-1 that
\[ E^S(a^q) \subseteq \langle s, a^q(s) + a^q(s)^+ \rangle \] and thereby that \[ \exists E^S(a^q) \otimes R_1^q(s, r'') \] as required.

It is easy to see that \[ s_2 \uparrow d' \] holds since \[ s_2 \uparrow \) and \[ d'(p') = 0 \] for 
\[ p \notin \text{PROC}(L_1^q). \] This proves the lemma. //

**Proof of the Completeness Result**

In order to prove The Completeness Theorem for \[ R^R \] we shall prove that for any trace list 1 and assumption list \[ \phi_1 \] of the form
\[ \phi_1 \langle \phi_1 \circ \phi_2 \circ \cdots \circ \phi_k \rangle \] (where 1 is \[ p_1 \cdots p_k \]) we have

237
if $\Phi \Rightarrow P(c:R)Q/(V_1, env, L_1)$ is a well-formed formula satisfying
that if $p$ is in $FP(c)$ then either $p$ is in $l$ or $lp$ is a trace
list and furthermore

\[(\$)\]

then

\[R^P\Phi \Rightarrow P(c:R)Q/(V_1, env, L_1).\]

For a moment assume that $l$ is the empty list. Then $\Phi$ is $\lambda$, $V_1$ is $V$ and $L_1$ is $\emptyset$. Since any list of procedure names of length one is a trace list we see that The Completeness Theorem for $R^P$ follows from ($\$)$.

The proof of ($\$)$ will be by induction on the "negative" length of $l$ and the structure of $c$. In order to define the "negative" length of a trace list $l$ we shall consider the set $EXT(l, env)$ of possible extensions of $l$ within $env$:

\[EXT(l, env) = \{l' | l' is a trace list over env\}.\]

This set will be finite since there are only a finite number of trace lists over a given environment - remember there are no repetitions of procedure names in the trace lists. The maximal length of a list in $EXT(l, env)$ will be the maximal extension of $l$ but only if no other procedures are declared. The "negative" length of $l$ is going to be a measure for how much $l$ can be extended. We shall therefore try to estimate the number of procedures that may be known at any point during the execution of a program.

For a program $c$ we define $depth(c)$, the depth of procedure declarations in $c$, as in the following table:

238
The definition of \( \text{ depth} (\text{PROC } p(\text{VAL } x). \text{ IS } c_1 \text{ IN } c_2) \) reflects that we during an execution of such a program may have information about both the procedures declared in \( c_1 \) and in \( c_2 \). Note that this is not the case for a program as \( c_1; c_2 \).

A list \( p_1 \ldots p_k \) of \( \text{EXT}(l, \text{env}) \) can at most give rise to an extension of \( l \) of length \( \text{len}(p_1 \ldots p_k) \) where

\[
\text{len}(p_1 \ldots p_k) = \text{depth}(c_1) + \ldots + \text{depth}(c_2) + k
\]

where \( \text{env}(p_i) = (x_i, c_i) \) for \( 1 \leq i \leq k \). Therefore, a trace list \( l \) can at most be extended by a list of length \( \text{neg}(l, \text{env}) \) where

\[
\text{neg}(l, \text{env}) = \max \{ \text{len}(l') \mid l' \text{ is in } \text{EXT}(l, \text{env}) \}.
\]

The "negative" length of \( l \) is thus defined to be the natural number \( \text{neg}(l, \text{env}) \). We shall adopt the convention that \( \max \emptyset = 0 \).

We shall say that the formula \( \Phi_1 \Rightarrow \exists c: R \diamond (V_1, \text{env}, L_1) \) satisfies the predicate \( \text{BOUND}(j) \) if it is well-formed and

- \( \Phi(c) \notin \text{EXT}(l, \text{env}) \cup \{ p \mid p \text{ is in } l \} \)
- \( \text{depth}(c) + \text{neg}(l, \text{env}) \leq j \).

The last condition reflects that the program \( c \) might declare some
procedures that also can be added to the trace list.

We shall first prove that (§) holds for formulas satisfying the predicate BOUND(0). Next we prove that if it holds for those satisfying BOUND(j) then it also holds for those satisfying BOUND(j+1). Since for any formula $\Phi_1 \Rightarrow P\langle c:R\rangle Q/(V_1, env, L_1)$ there is a $j$ such that BOUND($j$) holds we get that this will prove (§).

**BASIS**

Assume now that $\Phi_1 \Rightarrow P\langle c:R\rangle Q/(V_1, env, L_1)$ is a valid formula and that BOUND(0) holds. By structural induction on the program $c$ we shall construct a proof of the formula in $R$. The cases of $x:=e$, IF $b$ THEN $c_1$ ELSE $c_2$, WHILE $b$ DO $c$ and LET $x=e$ IN $c$ are modifications of those of the corresponding proofs in the completeness proof for the proof system $N$ in Section 4.4. The modifications are all due to the fact that we have added the assumption lists and are essentially the same for the five cases. We shall therefore only redo the proof for one of the cases, that of variable declaration. The case of procedure declaration cannot occur since we have assumed that BOUND(0) holds. So below we present the proofs for the cases LET $x=e$ IN $c$ and CALL $p(e)$.

**Case LET $x=e$ IN $c$:** Assume now that we have

(1) $\Phi_1 \Rightarrow P\langle LET x=e \rangle Q/(V_1, env, L_1)$

holds and we shall construct a proof of the formula in $R$. We proceed as in the completeness proof for $N$, case LET $x=e$ IN $c$, in Section 4.4 and define $Q'$ and $R'$ to be the formulas $\exists y. G_{V_1 \cup \{y\}}^{c_y}$, $env[\{c_x^y\}]$ and $E_{V_1 \cup \{y\}}^{c_y}$, $env[\{c_x^y\}]$ respectively, where $y$ is a variable of the same sort as $x$ satisfying $y \notin V_1$. Below we shall prove that the formula

240
(2) \( \Phi_1 \Rightarrow \text{PAY} = e(c^y_x : R') Q'/(V_1, u \{ y \}, \text{env}, L_1) \)

is valid. Since BOUND(0) holds for (1) it obviously also holds for (2) and we can apply the induction hypothesis and get a proof of (2) in \( R^R \). We can then apply the rule /LET-/ and get a proof of

\[ \Phi_1 \Rightarrow \text{PAY} = e(\text{LET } x = e \text{ IN } c : E(e) @ R', e) Q'/(V_1, u \{ y \}, \text{env}, L_1). \]

Below we prove that

(3) \( \Phi \text{PAY} = e \Rightarrow Q \)

and

(4) \( \Phi \text{PAY} = e \Rightarrow R. \)

So using first /inv-/ and then /cons-/ with the formulas (3) and (4) we get the proof of the required formula (1).

To prove that (2) is valid assume that for some counter \( d \)

\( d : L_1 \Phi_1,/(V_1, u \{ y \}, \text{env}) \)

holds for every prefix \( l' \) of \( l \). We shall prove that

(5) \( d : L_1 \Phi \text{PAY} = e \Rightarrow c^y_x : R' Q'/(V_1, u \{ y \}, \text{env}). \)

From Lemma 5.4-5 we have

\( d : L_1 \Phi_1,/(V_1, \text{env}) \)

for every prefix \( l' \) of \( l \). If therefore \( s \prec L_1 d \) and

\( \Phi \text{PAY} = e(s) \)

hold for some state \( s \) then the assumption (1) gives that

\( (V_1, \text{env}) \Rightarrow \text{LET } x = e \text{ IN } c, s \Rightarrow L_1 s' \)

for some \( s', r \) and \( d' \). This means, according to the semantic rule /LET-/ that for some \( r' \)

\( (V_1, u \{ y \}, \text{env}) \Rightarrow c^y_x, r' \text{PAY}(s') \Rightarrow L_1 s' \)

where \( y' \) is a variable of the same sort as \( x \) satisfying \( y' \not\in V_1 \).

241
Without loss of generality we can assume that \( y = y' \) (if it is not the case then the lemmas 5.1-2 and 5.1-5 can be used to perform the necessary renaming). Since \( y = e(s) \) holds we have \( s^e_y = s \). The definitions of \( Q' \) and \( R' \) together with the expressiveness assumption give directly that \( \forall Q'(s,s') = \) and \( \forall R'(s,r') \) hold and the validity of (2) follows.

To prove (3) assume that \( \forall Q'(s,s') \) holds for some pair \( (s,s') \) of states. This means that \( \forall P(s) \) holds. We can now define the depth counter \( d \) by

\[
d(p') = \begin{cases} 
  z'(p) & \text{if } (p',z') \in L_1 \\
  0 & \text{otherwise} 
\end{cases}
\]

We then have \( s \leq d \).

From Lemma 5.4-5 we have that \( d : L_1 \vdash (V_1, \text{env}) \) holds for every prefix \( l' \) of \( l \). So from (1) we get that for some \( r, d' \) and \( s'' \)

\[
(V_1, \text{env}) \vdash \langle \text{LET } x = e \text{ IN } c, s \rangle \Rightarrow d', s'' \text{ and } \forall Q(s,s'')
\]
since \( s \leq d \) holds. The rest of the proof showing that \( \forall Q(s,s') \) holds is as in the completeness proof for \( x \in \mathcal{L} \) in the case \( \text{LET } x = e \text{ IN } c \) and is therefore omitted here.

The proof showing that (4) holds is a modification of the proof for the similar result in Section 4.4 along the same lines as above. We omit the details.

Case \text{CALL } p(e): Assume that

\[
1) \quad \boxed{P \vdash (\text{CALL } p(e) : R \langle Q \rangle / (V_1, \text{env}, L_1)}
\]

and that \( \text{BOUND}(0) \) holds. We shall construct a proof of the formula in
Since $\text{BOUND}(0)$ holds it must be the case that $\text{EXT}(1,\text{env})=\emptyset$. But we also have $p \in \text{EXT}(1,\text{env}) \cup\{p'\}$ is in $1$ so we get that $p$ is in $1$.

Assume now that $1=1'p''$. The first step in the proof of (1) in $\mathfrak{A}^R$ is to apply the rule $/\text{sel-}\mathfrak{A}^R/$ and we get a proof of

$$\Phi \Rightarrow p_1p(z_p)(\text{CALL } p(a_p):E^S(a_p)@R_1p')_pQ_1p/(V_1,\text{env},L_1).$$

Using Lemma 5.4-4 it is straightforward to verify that this formula is well-formed and thereby that the application of the axiom is possible.

The rule $/\text{par-}\mathfrak{A}^R/$ can then be applied to replace the actual parameter $a_p$ of the call by the required term $e$. The formula $Q_1p'$ is constructed such that $a_p \notin \text{FV}(Q_1p')$ and since $a_p \notin \text{FV}(\text{env})$ (E) and $a_p \notin \text{FV}(p)$ we get a proof of

$$\Phi \Rightarrow p_1p(z_p)a_p \langle \text{CALL } e:E^S(e)@\langle R_1p'\rangle_p^eQ_1p\rangle_p^e/(V_1,\text{env},L_1).$$

The formula $Q_1p'$ expresses a property of the variables of $V_1p'-\text{VAR}(L_1)$. We shall now extend the post-condition to express a property of all the variables of $V_1p'-\text{VAR}(L_1)$. So define $A=\{p,p' \text{ is in } p''\}$. Then $A \subseteq V_1p'-\text{FV}(\text{env})\cup\text{VAR}(L_1)$ and we can apply the rule $/\text{ext-}\mathfrak{A}^R/$ and get a proof of

$$\Phi \Rightarrow p_1p(z_p)a_p \langle \text{CALL } e:E^S(e)@\langle R_1p'\rangle_p^eQ_1p\rangle_p^eA/(V_1,\text{env},L_1).$$

Below we shall prove that

\begin{align*}
(2) & \quad \Phi \Rightarrow p_1p(z_p)e_p, \\
(3) & \quad \Phi \Rightarrow Q_1p\langle e \wedge I \rangle_A \Rightarrow Q, \\
\end{align*}

and

\begin{align*}
(4) & \quad \Phi \Rightarrow E^S(e)@\langle R_1p'\rangle_p^eR \\
\end{align*}

So using first $/\text{cons-}\mathfrak{A}^R/$ with (2), then $/\text{inv-}\mathfrak{A}^R/$ and finally $/\text{cons-}\mathfrak{A}^R/$ with (3) and (4) we get the required
proof of (1) in $\mathcal{R}$.

To prove (2) assume that $\forall P(s)$ holds for some state $s$. We define the counter $d$ such that

$$d(p') = \begin{cases} z'(s) & \text{if } (p', z') \in L_1 \\ 0 & \text{otherwise.} \end{cases}$$

From Lemma 5.4-5 we get that

$$d \in L_1 \vdash \phi_0 / (V_1, \text{env})$$

holds for every prefix $1_0$ of $1$. From the assumption (1) we therefore get that

$$d \in L_1 \vdash P<\text{CALL } p(e): R> \text{Q} / (V_1, \text{env})$$

holds. Since $s \in d$ and $\forall P(s)$ hold this means that for some $s', r$ and $d'$

$$(V_1, \text{env}) \vdash \langle \text{CALL } p(e), s \rangle \xrightarrow{r} d', s' \text{ and } s \geq d'.$$

Since $a \in V_1 - \text{FV(} \text{env})$ we can apply Lemma 5.1-9 and replace the actual parameter $e$ by $a$

$$(V_1, \text{env}) \vdash \langle \text{CALL } p(a), s \rangle \xrightarrow{r} d', s'$$

for some $r'$ and $s'$. The next step is to reduce the set $V_1$ to $V_1$.

We have $(V_1 \setminus V_1, p) \cap (\text{FV(} \text{env}) \cup \text{FV(} \text{CALL } p(a), p)) = \emptyset$ since $a \in V_1$ and $p$.

Lemma 5.1-2 then gives

$$(V_1, p, \text{env}) \vdash \langle \text{CALL } p(a), s \rangle \xrightarrow{r'} d', s'$$

for some $s'$. Above we have $s \geq d'$ so therefore $s \geq d'$. We can apply Lemma 5.4-2 and replace env by $\text{env}_1$.

$$(V_1, p, \text{env}, \text{env}_1) \vdash \langle \text{CALL } p(a), s \rangle \xrightarrow{r''} d', s'$$

for some $s'$ and $r''$. The expressiveness assumption together with the definition of $P_1, p(z)$ gives that $\forall P_1, p(z) s^e(s)$ holds. But then $\forall P_1, p(z) s^e(s)$ holds and we have completed the proof of (2).
We now turn to the proof of (3) so assume that for some pair \((s, s')\) of states \(\langle P \wedge (Q_1', p) \rangle_{s, s'}\). Let us first see what can be deduced from \(\langle Q_1', p \rangle_{s, s'}\). The definition of \(Q_1', p\) and the expressiveness assumption give that

\[(V_1, p, -\text{VAR}(L_1, p), s) \vdash \langle \text{CALL } p(e), s \rangle_{a, p} \rightarrow_{d, s''} s''\]

for some \(s'', d, r\) satisfying that \(s'' \in V_1, p, -\text{VAR}(L_1, p)\). Using Lemma 5.1-3 we can extend the set \(V_1, p, -\text{VAR}(L_1, p)\) to \(V_1, -\text{VAR}(L_1)\) and thus we have

(5) \(\langle V_1, -\text{VAR}(L_1), s) \vdash \langle \text{CALL } p(e), s \rangle_{a, p} \rightarrow_{d, s''} s''\)

for some \(s''\) satisfying \(s'' \in V_1, p, -\text{VAR}(L_1, p)\) and \(s'' \in A\{a\}_{a, p}\) since \(V_1, p, -\text{VAR}(L_1, p) = A\{a\}_{a, p}\). From \(\langle I_1, s, s'\rangle\) and \(\langle V_1, -\text{VAR}(L_1, p)\rangle\) we get

(6) \(s'' \in V_1, -\text{VAR}(L_1)\).

From the other assumption \(\langle P, s, s'\rangle\) we get that \(P\) holds.

Furthermore we shall define a counter \(d_0\) as follows

\[d_0(p') = \begin{cases} z'(s) & \text{if } (p', z') \in L_1 \\ 0 & \text{otherwise.} \end{cases}\]

From Lemma 5.4-5 we have that

\[d_0 : L_1 \models \neg \phi_0 / (V_1, \text{env})\]

holds for every prefix \(l_0\) of \(l\). So from the assumption (1) we get

\[d_0 : L_1 \models \langle \text{CALL } p(e) : R \rangle_{Q, (V_1, \text{env})}.\]

Since \(s \in V_1, d_0\) and \(\models P(s)\) hold we get that

(7) \(\langle V_1, \text{env} \rangle \vdash \langle \text{CALL } p(e), s \rangle_{d, s'_{0}} \rightarrow_{d'} s_0'\) and \(\models Q(s, s_0')\)

for some \(s_0', d', r'\). Since \(a \in V_1, -\text{FV}(\text{env})\) we can apply Lemma 5.1-8.
and replace the actual parameter e by a_p:

\[(V_1, \text{env}) \vdash \langle \text{CALL } p(a_p), s \rangle_{a_p} \rightarrow^r_{d} s_1\]

where \(s_1 \in \text{V}_1 - \{a_p\} s_0\). We shall now replace the set \(V_1\) by \(V_1 - \text{VAR}(L_1)\).

We have \(\text{VAR}(L_1) \cap (\text{FV}(\text{env}) \cup \text{FV}(\text{CALL } p(a_p))) = \emptyset\) and since \(\text{FV}(e) \cap \text{VAR}(L_1) = \emptyset\), we also have \(s_1 \in V_1 - \text{VAR}(L_1)\) so Lemma 5.1-2 gives

\[(8) \quad (V_1 - \text{VAR}(L_1), \text{env}) \vdash \langle \text{CALL } p(a_p), s \rangle_{a_p} \rightarrow^r_{d} s_2\]

for some \(s_2\) satisfying \(s_1 \in V_1 - \text{VAR}(L_1) s_1\).

From (5) and (8) we now get that \(s_1 \in V_1 - \text{VAR}(L_1) s_1\) since the language is deterministic (Lemma 5.1-9). From above we have \(s_1 \in V_1 - \text{VAR}(L_1) s_0\) and (6) gives that \(s_1 \in V_1 - \text{VAR}(L_1) s_1\) so \(s_1 \in V_1 - \text{VAR}(L_1) - \{a_p\} s_0\). We shall below prove that \(a_p(s') = a_p(s')\) and thereby that \(s_1 \in V_1 - \text{VAR}(L_1) s_1\).

Since \(Q(s, s')\) and \(\text{FV}(Q) \subseteq (V_1 - \text{VAR}(L_1)) \cup (V_1 - \text{VAR}(L_1)) (1)\) is well-formed) we then get that \(\exists Q(s, s')\) as required.

To prove that \(a_p(s') = a_p(s')\) consider the formula (7). Let \(x'\) be a new variable of the same sort as \(x\), that is \(x' \in V_1\). The semantic rule \(/\text{CALL} - \text{VAR}/\) gives that

\[(V_1 \{x'\}, \text{env}) \vdash \langle x'_{x'}, s \rangle_{x'} \rightarrow^r_{d} s_0\]

where \(\text{env}(p) = (x, c)\) and for some \(r\) and \(d\). Since \(a_p(e(s))\) we have \(a_p(FV(x') \cup \text{FV}(\text{env}))\) and from Lemma 5.1-1 we get \(a_p(s, x') = a_p(s)\).

From \(\exists A(s, s')\) and \(a \in A\) we get \(a_p(s) = a_p(s')\) so \(a_p(s') = a_p(s')\). This completes the proof of (3).

Finally, we have to prove that (4) holds. So assume that for some pair \((s, r)\) of state and natural number \(P \in \text{VAR}(e) \cap (R_1 \cup \text{VAR}(L_1)) a_p(s, r)\).

Thus we have \(P(s)\) and as above we shall define the depth counter \(d_0\) such that \(s_1 \in V_1 d_0\). From the assumption (1) we then get
for some $s', r'$ and $d'$. The semantic rule $\text{CALL} \to \text{CALL}$ then gives that $r' = e'(s') + e(s') + r'$ for some $r'$ and that

$$(V_1 \cup x', \text{env}) \vdash \langle c, s, e(s) \rangle \to r', d', s'$$

for some $d''$ and for $x'$ being a new variable (that is, $x' \notin V_1$) of the same sort as $x$. We shall now replace the variable $x'$ by $a_p$. Since $a \notin \text{FV}(c'' \cup \text{FV}(\text{env}))$ we can remove $a$ from the set $V_1 \setminus x'$. Define $s_1 \xrightarrow{p} s$ to be the state $s \xrightarrow{a_p} s$. Then $s_1 \xrightarrow{p} s \in (V_1 \cup x')s \xrightarrow{a_p} s$ and Lemma 5.1-2 gives

$$((V_1 \cup x') \setminus \{a_p\}, \text{env}) \vdash \langle c, s, a_p(s) \rangle \to r', d', s_1$$

for some $s_1$. Using Lemma 5.1-5 we then get

$$(V_1 \cup x'), \text{env} \vdash \langle c, x', s, s_1 \rangle \to r', d', s'$$

for some $s'$. We can get rid of the variable $x'$ by applying Lemma 5.1-2 and since $(\{x'\} \setminus \text{VAR}(L_1)) - (\text{FV}(c'' \cup \text{FV}(\text{env})) = \emptyset$ and

$$s_1 \xrightarrow{p} s_1 \xrightarrow{a_p} s \xrightarrow{a_p} s$$

for some $s_1$. We also have that $\text{FR}(e) \otimes (R_{1,p} \otimes e(s,r))$. Using the time expressiveness assumption together with Lemma 3.2-1 we get that this means that $r = e'(s') + e(s') + r''$ for some $r''$ satisfying $\text{FR}(e) = (R_{1,p} \otimes e(s'), r'')$. The expressiveness assumption together with the definition of $R_{1,p}$ give us that

$$(V_1 \cup \text{VAR}(L_1), \text{env}) \vdash \langle c, s, e(s) \rangle \to r'', d'', s''$$

for some $s''$ and $d''$. Since $V_1 \cup \text{VAR}(L_1) \subseteq V_1 \cup \text{VAR}(L_1)$ we can apply Lemma 5.1-3 and get
for some $s''$. Combining this result with that of (9) gives us that 
$r''=r'_1$ since the language is deterministic (Lemma 5.1-9). But then 
$r=r'$ and from $R(s_0,r')$, $FV(R)\subseteq(V_1-VAR(L_1))$ ((1) is well-formed) and $s_0\subseteq(V_1-VAR(L_1))$ we get $F(R,s,r)$ as required. This com-
pletes the proof of (4) and thereby the basis step in the induction 
proving ($$).

THE INDUCTION STEP

Assume now that ($$) holds for any formula $P(c)\Rightarrow P(c')$ satisfying the predicate BOUND($j$) and we shall prove that it hold 
for the formulas satisfying BOUND($j+1$).

The proof is by structural induction on the program $c$. The cases 
of $x:=e$, IF $b$ THEN $c_1$ ELSE $c_2$, $c_1;c_2$, WHILE $b$ DO $c$ and LET $x=e$ IN $c$ 
are exactly as above in the basis case so we omit the details here. 
The cases of PROC $p(VAL x)$ IS $c_1$ IN $c_2$ and CALL $p(e)$ are considered 
below.

**Case PROC $p(VAL x)$ IS $c_1$ IN $c_2$:** Assume that 
(1) $P\Rightarrow P(\text{PROC} p(VAL x) \text{ IS } c_1 \text{ IN } c_2 ; R) \Rightarrow Q/(V_1, env, L_1)$

and that BOUND($j+1$) holds. We shall construct a proof of the formula 
in $\mathcal{R}$. Let $q$ be a new procedure name (that is, $q \notin \text{DOM}(\text{env})$) and define 
$\text{env'}$ to be the environment $\text{env}(q=(x,c_1))$. Then $l$ is a trace list over 
$\text{env'}$ and below we shall prove that the formula 
(2) $\Rightarrow P(c_2 ; R) \Rightarrow Q/(V_1, env', L_1)$

is valid. Below we shall also prove that the predicate BOUND($j+1$) holds 
for the formula. The induction hypothesis can then be applied and we
get a proof of (2) in $R^R$. But then the rule /PROC-RR/ can be applied and we get a proof of

$\Phi_1 \Rightarrow P<\text{PROC } p(\text{VAL } x) \text{ IS } c_1 \text{ IN } c_2 : R\neg Q/(V_1,\text{env},L_1)$

as required.

We have to prove that the formula (2) is valid. First we have to verify that the formula is well-formed. The sets $L_1$ and $V_1$ which are defined from the environment $\text{env}$ are the same as those defined from the environment $\text{env}'$. Therefore Lemma 5.4-4 gives that

$\Phi_1/(V_1,\text{env}')\text{PROC}(L_1))$ is well-formed relative to $L_1$ for any prefix $l'$ of $l$. From the well-formedness of the formula (1) it is straightforward to get that $P<\text{c}_{1:R\neg Q/(V_1,\text{env}')}\text{PROC}(L_1)$ is well-formed relative to $L_1$. This proves the well-formedness of (2). To prove that (2) is valid assume now that for some counter $d$ (over $\text{env}'$)

$d:L_1 \Phi_1/(V_1,\text{env}')$

holds for every prefix $l'$ of $l$. We have to prove that

(3) $d:L_1 \Phi_1 P<\text{PROC } p(\text{VAL } x) \text{ IS } c_1 \text{ IN } c_2 : R\neg Q/(V_1,\text{env}').$

Let now $d_0$ be the restriction of $d$ to $\text{DOM}(\text{env})$. Lemma 5.4-5 gives that

$d_0:L_1 \Phi_1/(V_1,\text{env})$

holds for every prefix $l'$ of $l$. So from (1) we get

(4) $d_0:L_1 \Phi_1 P<\text{PROC } p(\text{VAL } x) \text{ IS } c_1 \text{ IN } c_2 : R\neg Q/(V_1,\text{env}).$

To prove (3) assume now that $s \subseteq d_0$ and $\text{FP}(s)$ hold for some state $s$. Then $s \subseteq L_1 d_0$ and from (4) we get

$(V_1,\text{env}) \text{-PROC } p(\text{VAL } x) \text{ IS } c_1 \text{ IN } c_2 \text{ S}_0 \text{S}', \text{FP}(s,s'), \text{FR}(s,r)$

and $s_0 \subseteq L_1 d_0$. 

249
for some $s'$, $r$ and $d'_0$. The semantic rule \( /proc-q^{r'}/ \) then gives that

\[(5) \quad (v_1, env(q'=(x,c_{1p}^{q'}))) \vdash \langle c_{2p}^{q'}, s \rangle \rightarrow_d s', \]

for some procedure name $q'$ satisfying $q' \notin DOM(env)$ and where $d'$ is some extension of $d'_0$, that is, $d'(p') = d'_0(p')$ for $p' \in DOM(env)$. If $q' \notin q$ we shall rename $q'$ to $q$ using Lemma 5.1-6 so without loss of generality we can assume that $q' \in q$ in (5). We have $s \gg_d d'_0$ and since $q \in PROC(L_1)$ we get $s \gg_d d'$. This proves (3) and thereby (2).

We have to prove that $BOUND(j+1)$ holds for the formula (2) in order to apply the induction hypothesis. As we have seen above the formula is well-formed. Since $BOUND(j+1)$ holds for (1) we have

\[
FP(PROC p(VAL x) IS c_1 IN c_2) \subseteq EXT(1, env) \cup \{p'|p' \text{ is in } l\}. \]

If $p \notin FP(c_2)$ we get

\[
\forall q \in \{q\} \cup EXT(1, env) \cup \{p'|p' \text{ is in } l\}, \quad \forall c_{1p}^{q}, \quad c_{2p}^{q} \in FP(PROC p(VAL x) IS c_1 IN c_2) \subseteq EXT(1, env(q=(x,c_{1p}^{q}))), \quad \{p'|p' \text{ is in } l\}.
\]

The same inclusion holds if $p \notin FP(c_2)$. Since $BOUND(j+1)$ holds for (1) we have $depth(PROC p(VAL x) IS c_1 IN c_2) + neg(1, env) \leq j+1$. Let now $j_1 = depth(c_1)$ and $j_2 = depth(c_2)$; then $j_1 + j_2 + 1 \leq j+1$. For any $l'$ of $EXT(1, env)$ we have $len(l') \leq (j_1 + j_2 + 1)$. If $l''$ is in $EXT(1, env')$ then either $l''$ is in $EXT(1, env)$ or $l''$ "with $q$ erased" will be in $EXT(1, env)$. Therefore $len(l'') \leq ((j+1) - (j_1 + j_2 + 1)) + j_1 + 1 = j_2 + 1$. We thus get

\[
depth(c_{2p}^{q}) + neg(1, env') \leq j_2 + (j_2 + 1) = j+1.
\]

This proves that $BOUND(j+1)$ holds and thereby we have completed the proof in the case of procedure declarations.

Case CALL $p(e)$: Assume now that we have

---

250
and furthermore that BOUND(j+1) holds. We shall construct a proof of the formula in $\mathfrak{K}$. We have two cases depending on whether or not $p$ is in the trace list $l$.

In the case where $p$ is in $l$ we proceed exactly as in the basis step of the induction proving ($\dagger$), case CALL $p(e)$. (In that proof we only use the assumption that $l$ cannot be extended to conclude that $p$ is in $l$ so the proof can be adopted directly.)

Assume now that $p$ is not in $l$. Then FP(CALL $p(e)) \subseteq$ EXT($l,\text{env}$) so we get that $lp$ is a trace list. Below we shall prove that the following formula is valid

$$(2) \quad \Phi_{lp} \Rightarrow P_p(z+1)\langle c_p^{P,R}:l_p \rangle Q_p(l_p,\text{env},L_{lp})$$

where $\text{env}(p)=(x,c)$. It is straightforward to verify that it is a well-formed formula using Lemma 5.4-4. By construction of EXT($l_p,\text{env}$) we have FP($c_p^{P}) \subseteq$ EXT($l_p,\text{env}$)$\cup\{p' | p'$ is in $l_p\}$. Since BOUND(j+1) holds for (1) we have depth(CALL $p(e)) + \text{neg}(l,\text{env})j+1$. This means that

len($l'\text{p'})j+1$ for every $p'$ of EXT($l,\text{env}$). Let now $j'=$depth($c_p^{P})$. Then

len($l') < j'-j$ for every $l'$ of EXT($l_p,\text{env}$) (=\{l'' | p'' \subseteq$ EXT($l,\text{env}$)). So we have

depth($c_p^{P}) + \text{neg}(l_p,\text{env})j' + (j-j') (=j)$.

This proves that BOUND($j$) holds for (2) and the induction hypothesis gives us a proof of (2) in $\mathfrak{K}$. Below we prove that

$$(3) \quad \Phi_p \Rightarrow L_{lp}(0)$$

holds. We can then apply the rule $\text{CALL-} \mathfrak{K}$/ and get a proof of

$$(4) \quad \Phi_1 \Rightarrow P_p(z)_{ap} \langle \text{CALL } p(e):E^p(e)\otimes (R_{lp})_{ap} \rangle_{ap} Q_p(l_p,\text{env},L_{lp})$$
We now have two cases. If \( p \in \text{PROC}(L_1) \) then \( L_{1p} = L_1 \) and \( V_{1p} \setminus \{a_p\} = V_1 \).

Below we prove that

\[
\begin{align*}
(5) \quad & \chi_p \rightarrow p_{1p} (z)_{1p} a_p \\
(6) \quad & \chi_{\text{proc}} (Q_{1p})_{1p} a_p \\
\end{align*}
\]

and

\[
(7) \quad \chi_{\text{proc}} (e)(R_{1p})_{1p} a_p \rightarrow R.
\]

Using the rules \( \text{cons}-\mathcal{R} \) and \( \text{inv}-\mathcal{R} \) we therefore get a proof of the required formula (1).

If, on the other hand, \( p \notin \text{PROC}(L_1) \) we proceed as follows. Assume that \( L_1 = \{(p_1, z_1), \ldots, (p_k, z_k)\} \) and \( L_{1p} = L_1 \cup \{(p_{k+1}, z_{k+1}), \ldots, (p_{k+m}, z_{k+m})\} \).

We then have \( V_{1p} \setminus \{a_p\} = V_1 \cup \{z_{k+1}, \ldots, z_{k+m}\} \). We shall apply the rule \( \text{elim-}\mathcal{R} \) to the proof of (4) and get a proof of

\[
\chi_{1p} \rightarrow \exists z_{k+1} \ldots \exists z_{k+m}, p_{1p} (z)_{1p} a_p \langle \text{CALL } (e): E^S (e)(R_{1p})_{1p} a_p \rangle (Q_{1p})_{1p} a_p \rightarrow (V_1, \text{env}, L_1).
\]

In order for this to be a valid application of the rule \( \text{elim-}\mathcal{R} \) the conclusion of the rule must be well-formed but this is straightforward to verify using Lemma 5.4-4 and that \( V_{1p} \setminus \text{VAR}(L_{1p}) = V_1 \cup \{a_p\} \).

Below we shall prove that

\[
(8) \quad \xi_p \rightarrow (\exists z_{k+1} \ldots \exists z_{k+m}, p_{1p} (z)_{1p} a_p)
\]

so using the rules \( \text{cons}-\mathcal{R} \) and \( \text{inv}-\mathcal{R} \) together with (8), (6) and (7) we get the required proof of (1) in \( \mathcal{R} \).

We have to prove that the formula (2) is valid. So let \( d \) be a counter and assume that

\[
d: L_{1p} \xi_{1p}/(V_{1p}, \text{env})
\]

252
holds for every prefix $1'$ of $lp$. We have to prove that
\[ d : Lp \vdash lp(z + 1) \langle c^p_x : Rlp \rangle \sigma_{lp} / (V_{1p}, env). \]

So assume that $s \in Lp \vdash d$ and $lp(z + 1)(s)$ hold for some state $s$. Then
\[ z(s) + 1 \vdash lp(s) \]
and from the expressiveness assumption and the definition of $lp(z)$ we get
\[ (V_{1p}, env_{1p}) \vdash \langle CALL p(a_p), s \rangle_{z_p} \rightarrow_{d, s'} \]
for some $s'$, $r$ and $d'$. From Lemma 5.4-3 we get that $z(s) + 1 \vdash c^{p(s) + 1}_{L_p}$
and
\[ (V_{1p}, env_{1p}) \vdash \langle CALL p(a_p), s \rangle_{z_p} \rightarrow_{d, s'_1} \]
for some $s'_1$ and $r'$. Since $z \in FV(CALL p(a_p)) \cup FV(env)$ we can first apply
Lemma 5.1-2 and then Lemma 5.1-3 and get
\[ (9) \ (V_{1p}, env_{1p}) \vdash \langle CALL p(a_p), s \rangle_{x'_{1}}, \]
for some $s'_{2}$. Using the semantic rule $/CALL-\mathbb{R}/$ we get that for some
$r''$ and $d''$
\[ (V_{1p}, u \{x'\}, env_{1p}) \vdash \langle c^x_{x', s_{x'}} \rangle_{x, a_p(s)} \rightarrow_{d'', s''} \]
where $x'$ is a variable of the same sort as $x$ satisfying $x' \not\in FV_{1p}$ and
$d''=d'(p=d'(p)-1)$. We can now rename $x'$ to $a_p$ as follows. First we use
Lemma 5.1-2 to remove $a_p$ from the set $V_{1p} \cup \{x'\}$:
\[ ((V_{1p}, u \{x'\}) - \{a_p\}, env_{1p}) \vdash \langle c^x_{x', s_{x'}} \rangle_{x, a_p(s)} \rightarrow_{d'', s''} \]
where $s'_3 \in V_{1p} - \{a_p\}$. Then Lemma 5.1-5 gives:
\[ (V_{1p}, u \{x'\}, env_{x'}) \vdash \langle c^x_{x', s_{x'}} \rangle_{x, a_p(s)} \rightarrow_{d'', s''} \]
where $s'_4 \in V_{1p} - \{a_p\}$. And using Lemma 5.1-2 we now get
\[ (10) (V_{1p}, env_{1p}) \vdash \langle c^p_{x}, s \rangle_{x, a_p(s)} \rightarrow_{d'', s''} \]
where $s'_5 \in V_{1p}$. 

253
To complete the proof of (2) we have to prove that $\exists_{R_1p} (s, s')$ and $s^2 \sim d^u$. Since $\text{VAR}(L_1) \cap (FV(CALL p(a)) \cup FV(env)) = \emptyset$, we get from (9) and Lemma 5.1-2 that

$$(V_1 \cdot \text{VAR}(L_1), env) \vdash \langle \text{CALL } p(a), s \rangle_{\rightarrow d, s'_1}$$

for some $s'_1$ satisfying $s^u \equiv_{V_1 \cdot \text{VAR}(L_1)} s'_2$. From the deductions above we have $s^u \equiv_{V_1 \cdot \text{VAR}(L_1)} (\text{VAR}(L_1) \cup \text{env})^s_5$. The expressiveness assumption together with the definition of $Q_{1p}$ therefore gives $\exists_{Q_{1p}} (s, s')$.

The expressiveness assumption and the definition of $R_{1p}$ give that $\exists_{R_{1p}} (s, s')$ holds since Lemma 5.1-2 applied to (10) gives

$$(V_1 \cdot \text{VAR}(L_1), env) \vdash \langle \text{CALL } p(a), s \rangle_{\rightarrow d, s'_6}$$

for some $s'_6$. As we have seen above $s^u \equiv_{V_1 \cdot \text{VAR}(L_1)} s'_6$. The semantic rule $\text{CALL}$/ then gives (for some $x'$, $x' \equiv_{V_1 \cdot \text{VAR}(L_1)}$)

$$(V_1 \cdot \text{VAR}(L_1), env) \vdash \langle \text{CALL } p(a), s \rangle_{\rightarrow r, d'}$$

for some $s$, $r$ and $d$. The semantic rule $\text{CALL}$/ then gives (for some $x'$, $x' \equiv_{V_1 \cdot \text{VAR}(L_1)}$)

$$(V_1 \cdot \text{VAR}(L_1), env) \vdash \langle \text{CALL } p(a), s \rangle_{\rightarrow r, d'}$$

for some $r$ and $d'$. Remember now that $c$ is the program

$$\text{IF } 0 < z \text{ THEN } (z := z - 1; c; z := z + 1) \text{ ELSE loop.}$$

So the semantic rule $\text{IF}$/ gives that

$$(V_1 \cdot \text{VAR}(L_1), env) \vdash \langle \text{IF } 0 < z \text{ THEN } (z := z - 1; c; z := z + 1) \text{ ELSE loop, s} \rangle_{\rightarrow r, d', s'}$$

for some $r''$. But this is a contradiction since the program loop never
terminates. So it cannot be the case that $P_{P}(0)(s)$ holds for some state $s$. This proves (3).

The proof of (5), $P \rightarrow_{P_{P}}(z_{e})_{e}$, is essentially as that proving the similar result in the proof of the basis of the induction proving (5), case CALL p(e). So we shall omit the details here.

We now turn to the proof of the formula (6): $P \wedge (Q_{P})_{e} \rightarrow Q$. The proof is a simplification of the proof of the similar result in the basis case above, case CALL p(e), where we prove

$$P \wedge (Q_{P})_{e} \rightarrow Q.$$ 

We omit the details.

The proof showing that the formula (7): $P \wedge (P_{S}(e) \theta (R_{P})_{e}) \rightarrow R$, holds is a simple modification of the proof of a similar result in the basis, case CALL p(e). Again we omit the details.

Finally, we have to prove that (8), $P \rightarrow (\exists z_{1} \ldots \exists z_{k+m} P_{P}(z_{e})_{e})$ holds. The proof of this is a simple modification of that proving that (5) holds so we omit the details. This completes the proof in the case CALL p(e).

This completes the proof of the induction step in the proof of (5) and thereby The Completeness Theorem for $\mathcal{X}^R$.

5.5 EXAMPLES: FACTORIAL AND MERGE SORTING

In order to discuss the pragmatic issues concerning the proof system $\mathcal{X}^R$ we shall in this section consider two worked examples. As for the proof systems of the previous chapters the important question is how naturally we can formalise the traditional informal analyses.
For programs involving recursive procedures it is especially interesting to see how close we can come to the recurrence relations of the informal analyses.

In Chapter 4 we distinguished between three different proof strategies for applying a proof system as $\mathcal{R}$ to analyse the run-time of programs. The two of the strategies are the traditional ones: we can construct the proof in a bottom-up manner or in a top-down manner. In the third approach the idea is to introduce "unknown" time formulas and construct a "proof" of some property of the run-time of the program together with a list of conditions on the "unknown time formulas". It seems that especially this proof strategy is useful when formalising the informal analysis of recursive procedure programs. We shall illustrate this by the recursive factorial program that we already have considered in Section 5.2.

The second example of this section is the well-known merge sort algorithm and we shall demonstrate how we can obtain a rather close formalisation of the traditional informal analysis of this algorithm.

**THE FACTORIAL PROGRAM**

In Section 5.2 we considered the following recursive version of the factorial program:

```plaintext
PROC fac(VAL x)
IS  IF x=0 THEN res:=1 ELSE (CALL fac(x-1); res:=x*res)
IN  CALL fac(n)
```

Using the proof system $\mathcal{R}$ we proved that the run-time of this algorithm is $11' n+7$. A combined bottom-up/top-down strategy was used in the proof.
This formal analysis differs from the traditional one in that we (apparently) do not construct a recurrence relation. The informal analysis will give rise to the recurrence relation

\[
\begin{align*}
T(0) &= 6 \\
T(x+1) &= 11 + T(x) \quad \text{for } x \geq 0
\end{align*}
\]

where \( T(x) \) is the time required for the call CALL \( \text{fac}(x) \) (except that we must add the time required to evaluate the actual parameter \( x \)). A solution to (\$) is \( T(x) = 11x + 6 \).

Although the formal proof of Section 5.2 does not give rise to a recurrence relation as (\$) there are, never the less, deductions that vaguely resemble checking that some formula is a solution to some recurrence relation. When analysing the branches of the conditional we use that

1. \( \Phi(z+1) \land a = 0 \land \text{time} = 2 \Rightarrow \text{time} = 11'a + 2 \)

and

2. \( \Phi(\text{time} = 4) \land (\text{time} = 11'a + 5) \land a = 1 \land (\text{true} \land (\text{time} = 4)) \Rightarrow \text{time} = 11'a + 2 \).

This can be viewed as expressing that \( 11'a + 2 \) is a solution to the recurrence relation

\[
\begin{align*}
T'(0) &= 2 \\
T'(a) &= 11 + T'(a-1) \quad \text{for } a \geq 1.
\end{align*}
\]

We shall now show that by imposing another proof strategy we can make this more explicit in the formal proof.

**REDOING THE FORMAL PROOF**

We shall look for conditions on an unknown term \( F(n) \) that will ensure that we have a proof of the formula.
\((\$$)\quad \text{TRUE}(\text{program:time}=F(n)+1) \rightarrow \text{TRUE}/\{\{n, \text{fac}\},(),(),0\}.

Intuitively, \(F(n)\) is a term (with \(n\) as the only free variable) for the run-time of the call CALL \(\text{fac}(n)\) when the parameter \(n\) has been evaluated, that is, \(F(n)\) corresponds to \(T(n)\) in the informal analysis.

The proof of \((\$$)\) proceeds very much as in Section 5.2. So for the main program we get that it is sufficient to prove the formula

\[
P(z)^n_{\text{a}} \langle \text{CALL fac}(n): \text{time} = F(n)+1 \rangle \text{TRUE}/\{\{n, \text{res}, z\}, \text{env}, \{(\text{fac}, z)\}\}
\]

where \(\text{env}\) is the environment containing the declaration of the procedure \(\text{fac}\) and \(P(z)\) is the formula \(0 < z = a + 1\). The assumption list is then extended to contain the formula

\[
P(z) \langle \text{CALL fac}(a): \text{time} = 2) \#(\text{time} = F(a)-1) \rangle \text{TRUE}
\]

(abbreviated \(\phi'\)) and we shall construct a proof of the formula

\[
\phi' \Rightarrow P(z+1) \langle \text{IF a}=0 \text{ THEN res}:=1 \text{ ELSE (CALL fac}(a-1); \text{res}:=a'; \text{res}): \\
\hspace{1cm} \text{time} = F(a)-1 \rangle \text{TRUE}/\{\{n, \text{res}, z, a\}, \text{env}, \{(\text{fac}, z)\}\}.
\]

To prove this formula for the procedure body it is sufficient to prove the two formulas

\[
\phi' \Rightarrow P(z+1) \& a=0 \langle \text{res}:=1; \text{time} = F(a)-4 \rangle \text{TRUE}/\{\{n, \text{res}, z, a\}, \text{env}, \{(\text{fac}, z)\}\}
\]

and

\[
\phi' \Rightarrow P(z+1) \& a=0 \langle \text{CALL fac}(a-1); \text{res}:=a'; \text{res}; \text{time} = F(a)-4 \rangle \text{TRUE}/
\hspace{1cm} \{(n, \text{res}, z, a), \text{env}, \{(\text{fac}, z)\}\}
\]

(since \(E(a=0)\) can be chosen to be \(\text{time}=3\)). We can easily get a proof of the first of these formulas if the condition

\[
(3) \quad P(z+1) \& a=0 \& \text{time} = 2 \rightarrow \text{time} = F(a)-4
\]

is fulfilled. The second formula can be proved if the condition
is fulfilled.

The two conditions (3) and (4) specify some sort of recurrence relation. It is easy to see that it is sufficient to find a term $F(a)$ satisfying

$$F(0) = 6$$

and

$$F(a) = 11 + F(a-1)$$ for $a \geq 0$.

But this is exactly the recurrence relation ($) from the informal analysis.

This suggests that the proof strategy using "unknown time formulas" should be preferred when analysing recursive procedures. This is not surprising since it is also one of the ideas behind the informal analysis of recursive procedures: "What we must now do is to associate with each recursive procedure an unknown time function $T(n)$ ..." (/AHU82 p24/).

**Merge Sorting**

We shall now show how the run-time of the merge sort algorithm can be analysed in the proof system $\mathcal{R}$. Using the data type of one-dimensional arrays of Example 3.1.2 the algorithm can be written as follows - note the variable 1 of sort array contains the array to be sorted.
The idea in the algorithm is that in order to sort the sublist
\(1[i+1], \ldots, 1[i+m]\) of the array \(1\) it is sufficient to sort the two
sublists \(1[i+1], \ldots, 1[i+m/2]\) and \(1[i+m/2+1], \ldots, 1[i+m]\) and then
merge these two sorted sublists. The procedure merge will do that by
first copying the \(m\) elements into another array, \(copy\), and then put
them back into \(1\) in the correct order.

THE INFORMAL RUN-TIME ANALYSIS

An informal analysis of the algorithm was presented already in
Section 1.3. At that time we had no fixed idea of the exact cost of
the various operations, so we shall here repeat the arguments using
the uniform computational model for the data type given in Example
3.1-3.
The term $T(m)$ is going to be an upper bound on the run-time of a call of the procedure merge-sort with the parameters $i$ and $m$ - it is convenient to assume that they already have been evaluated. We shall now explain how we arrive at the recurrence relation

\[
T(1) = 5 \\
T(m) \geq 2T(m/2) + 50(m+1) \text{ for } m > 1.
\]

Consider first the true branch of the conditional of the body of the procedure merge. The test $m=1$ requires three units of time and the assignment $m:=1$ two units so we get $T(1)=5$. For the false branch we still get three time units from the test $m=1$. By assumption each of the two recursive calls requires at most $T(m/2)$ time units, however, we have to account separately for the evaluation of the parameters: the first call requires six time units and the second ten. We now have to argue that the procedure merge requires at most $50m+31$ time units. First we note that the evaluation of the parameters take six units and since the procedure is called with the parameter $m/2$ it is sufficient to argue that the body of the procedure requires less than $100m+25$ time units. A careful checking (which we shall omit) shows that this is indeed the case.

**RUN-TIME ANALYSIS IN $\mathbb{R}^R$**

We shall now turn to a formal analysis of the algorithm in the proof system $\mathbb{R}^R$. We shall use the proof strategy with the unknown time formulas as it was successful when analysing the factorial program earlier. So we shall look for conditions on the term $M(n)$ that will give us a proof of the formula

\[(1) \quad \forall \phi_0 [\text{length}(l) = n \langle \text{program:time} \langle M(n)+6 \rangle \text{TRUE}/\langle \{1,n\},(),0 \rangle] \]

in $\mathbb{R}^R$. As in the informal analysis the idea is that $M(m)$ is the time
required for a call of the procedure \textit{merge-sort} when its parameters have been evaluated.

Let us first consider a proof for the main program. From the rule \textit{/PROC-\textit{R}}, we get that in order to prove (1) it is sufficient to prove
\[ \lambda \forall n. \text{length}(l) = n \Rightarrow \text{CALL merge-sort}(1, \text{length}(l)) : \text{time}(M(n) + 6) \leftrightarrow \text{TRUE}/(\{1, n\}, \text{env}, \emptyset) \]
where \text{env} is the environment declaring the procedure \textit{merge-sort}. We shall now choose the invariant, \(P(z)\), for the procedure \textit{merge-sort} to be the formula \(0 < z < 2^n \land m \leq 2^Z\). Then we have
\[ \forall n. \text{length}(l) = n \Rightarrow \exists z. P(z) \land \text{length}(l) \]
so using \textit{/cons-\textit{R}} and \textit{/inv-\textit{R}} we get that it is sufficient to prove that
\[ (2) \lambda \exists z. P(z) \land \text{length}(l) \Rightarrow \text{CALL merge-sort}(1, \text{length}(l)) : \text{time}(M(m) + 6) \leftrightarrow \text{TRUE}/(\{1, n, z\}, \text{env}, L) \]
where \(L = \{(\text{merge-sort}, z)\}\). Let now \(\phi\) be an abbreviation for the formula
\[ P(z) \land \text{CALL merge-sort}(i, m) : \text{time}(=4) \land \text{time}(M(m)) \leftrightarrow \text{TRUE}. \]
This reflects that \(E^S(i) \land E^S(m)\) is equivalent to \text{time} = 4 and that \(M(m)\) is meant to be an upper bound on the run-time requirements of the procedure when the parameters have been evaluated. Since \(\forall z. P(0)\) holds, \(E^S(1)\) is equivalent to \text{time} = 2 and \(E^S(\text{length}(l))\) is equivalent to \text{time} = 6 we can get a proof of (2) by applying (an appropriate version of the rule \textit{/CALL-\textit{R}}) to a proof of the formula
\[ \phi \Rightarrow P(z+1) \land \text{PROC merge(\ldots) IS \ldots IN \ldots : time}(M(m)) \leftrightarrow \text{TRUE}/(V, \text{env}, L) \]
where \(V = \{1, n, z, i, m\}\).

We now turn to a proof of this property for the procedure \textit{merge}.
Let env' be the extension of the environment env that also declares the procedure merge. Using the rules /PROC-R/ and /IF-R/ we get that it is sufficient to prove the following two formulas in \( \mathcal{A}^R \):

(3) \( \phi \Rightarrow P(z+1) \land m=1 \land \text{time} \leq M(m) - 3 \Rightarrow \text{TRUE} / (V, \text{env}', L) \)

and

(4) \( \phi \Rightarrow P(z+1) \land m=1 \land \text{CALL merge-sort}(i, m/2); \text{CALL merge-sort}(i+m/2, m/2); \text{CALL merge}(i, m/2); \text{time} \leq M(m) - 3 \Rightarrow \text{TRUE} / (V, \text{env}', L) \).

We have here used that \( E(m=1) \) can be chosen to be \( \text{time}=3 \).

Using /ass-R/, /inv-R/ and /cons-R/ it is straightforward to construct a proof of (3) provided that \( M(m) \) satisfies

\[ P(z+1) \land m=1 \land \text{time} \leq 2 \Rightarrow \text{time} \leq M(m) - 3 \]

(remember, \( E^S(1) \) is equivalent to \( \text{time}=2 \)).

The proof of (4) is more complicated. From the axiom /sel-R/ we get a proof of the formula

\[ \phi \Rightarrow P(z) \langle \text{CALL merge-sort}(i, m); \text{time} \leq 4 \rangle \Rightarrow \text{time} \leq M(m) \rangle \Rightarrow \text{TRUE} / (V, \text{env}', L). \]

Using the rule /par-R/ we can replace the actual parameter \( m \) of the call by \( m/2 \) and we thus have a proof of

\[ \phi \Rightarrow P(z) \langle \text{CALL merge-sort}(i, m/2); \text{time} \leq 6 \rangle \Rightarrow \text{time} \leq M(m/2) \rangle \Rightarrow \text{TRUE} / (V, \text{env}', L) \]

since \( E^S(i) \otimes E^S(m/2) \) is equivalent to \( \text{time}=6 \). Because \( m \not\in \text{FV}(\text{env}') \) we can apply /ext-R/ and get a proof of

\[ \phi \Rightarrow P(z) \langle \text{CALL merge-sort}(i, m/2); \text{time} \leq 6 \rangle \Rightarrow \text{time} \leq M(m/2) \rangle \Rightarrow m=m \land z=z / (V, \text{env}', L) \]

We can now use that
\[ P(z+1) \wedge \neg m = 1 \rightarrow P(z)_{m/2} \]

and

\[ \phi P(z)_{m/2} \wedge m = m \wedge z = z \rightarrow P(z)_{m/2} \wedge m = m \]

so the rules \( /\text{cons-}R^R/ \) and \( /\text{inv-}R^R/ \) give a proof of

1. \( \phi \Rightarrow P(z+1) \wedge \neg m = 1 \langle \text{CALL merge-sort}(i,m/2); \text{time}(M(m/2)+6) \rangle \)
\[ P(z)_{m/2} \wedge m = m / (V, env', L). \]

In a similar way we can construct a proof of the formula

2. \( \phi \Rightarrow P(z)_{m/2} \langle \text{CALL merge-sort}(i+m/2,m/2); \text{time}(M(m/2)+10) \rangle \)
\[ m = m / (V, env', L). \]

Assume for a moment that we have a proof of the formula

3. \( \phi \Rightarrow \text{TRUE} \langle \text{CALL merge}(i,m/2); \text{time}(50\cdot m+31) \rangle \text{TRUE} / (V, env', L). \)

We can now put the proofs of (5), (6) and (7) together using \( /; R^R/ \)
and we get a proof of the formula

\[ \phi \Rightarrow P(z+1) \wedge \neg m = 1 \langle \text{CALL merge-sort}(i,m/2); \text{CALL merge-sort}(i+m/2,m/2); \text{CALL merge}(i,m/2); \text{time}(M(m/2)+6) \rangle \]
\[ (\text{time}(M(m/2)+10) \rangle \rightarrow \text{time}(2\cdot M(m/2)+50\cdot m+47). \]

Provided that

\[ P(z+1) \wedge \neg m = 1 \wedge \text{time}(2\cdot M(m/2)+50\cdot m+47) \rightarrow \text{time}(M(m)-3) \]

holds we therefore get a proof of (4) using \( /\text{inv-}R^R/ \) and \( /\text{cons-}R^R/ \).

In order to complete the proof of (4) we have to construct a proof
of the formula (7) in $X^R$. The proof is fairly straightforward and since it does not provide further insight we shall omit the details here.

To summarise, we have obtained a proof of the upper bound $M(n) + 6$ on the run-time of the merge sort algorithm provided that the two conditions

$$P(z+1)^A m=1 \land \text{time}=2 \rightarrow \text{time} \leq M(m) - 3$$

and

$$P(z+1)^A m=1 \land \text{time}(2 \cdot M(m/2) + 50 \cdot m + 47) \rightarrow \text{time} \leq M(m) - 3$$

are fulfilled. These conditions are fulfilled if and only if $M(n)$ satisfies the conditions

$$M(1) \geq 5,$$

$$M(m) \geq 2 \cdot M(m/2) + 50 \cdot (m+1) \quad \text{for } m \geq 1.$$ 

Except for the requirement $M(1) \geq 5$ (rather than $M(1)=5$) this is exactly the recurrence relation of the informal analysis.

5.6 CONCLUDING REMARKS

We have in this chapter seen how the proof system for run-time analysis of non-recursive procedure programs can be modified such that it also applies to programs with recursive procedures. The new proof system $X^R$ has been proved to be sound and complete in the same sense as the previous proof system. Based on worked examples we argue that we can obtain natural formalisations of the traditional informal analyses in that we obtain recurrence relations in the formal proofs that are close to those of the informal analyses.

The idea of counting the recursion depth in the specification of
the semantics of the recursive procedure language in Section 5.1 occurs in several papers constructing proof systems for recursive procedures. In for instance /Ap81/ it is accomplished by defining successive approximations to programs involving recursive procedure calls. The approximating programs are obtained by syntactic substitutions and for a procedure p with the formal parameter x and the body c /Ap81/ will define the approximating programs c_0, c_1, ... by

\[
\begin{align*}
  c_0 & \equiv \text{loop} \\
  c_{i+1} & \equiv c_p^i
\end{align*}
\]

where \( c_p^i \) is the program obtained by replacing the calls of p, CALL p(e), by the statement LET \( x=e \) IN \( c_i \). It is then straightforward to verify that if \( (V,(p=(x,c))) \triangleright (c_i,s) \triangleright d,s' \) then \( d(p) \triangleright i \). We note that /Ap81/ only consider the case where a single procedure is declared whereas we consider the more general case with nested declarations of procedures. This also explains why we use the more complicated structure of a depth counter rather than a single natural number.

The proof system \( \mathcal{K}^R \) of Section 5.2 defines (indirectly) a proof system \( \mathcal{T}^R \) for proving total correctness of programs in the recursive procedure language. The formulas of \( \mathcal{T}^R \) have the form

\[
P_1 \langle c_i \rangle Q_1 \& ... \& P_k \langle c_k \rangle Q_k \Rightarrow P_0 \langle c_0 \rangle Q_0 / (V, \text{env}, L)
\]

where \( P_i, c_i, Q_i, V, \text{env} \) and L are as in Section 5.2 (for \( 0 \leq i \leq k \)). The axioms and rules of the proof system \( \mathcal{T}^R \) are obtained from those of \( \mathcal{K}^R \) by removing the deductions about time formulas. The axioms and rules of \( \mathcal{T}^R \) are thus the straightforward extensions of those of the proof system \( \mathcal{T}^N \) (except /CALL-\( \mathcal{T}^N \)/) considered in Section 4.6 together with the following axioms and rules: (\( \emptyset \) is now a possible empty sequence of formulas of the form \( P(c)Q \)):
\[ \phi \supset \text{CALL } \text{p(a)} \supset Q \supset \text{p(z+1)} \supset \text{Q(x)} / (V \cup \{a\}, \text{env}, L), \quad \neg \text{p(0)} \]

where \( \text{env}(p) = (x, c), \ (p, z) \in \text{L}, \ z \) is a variable of sort \text{nat} satisfying \( z \notin \text{FV}(Q) \) and \( a \) is a variable of the same sort as \( x \) satisfying \( a \notin \text{FV}(Q) \).

\[ \phi \Rightarrow \text{P}(c \supset Q) / (V, \text{env}, L) \]

where \( \text{P}(c \supset Q) \) is in \( \phi \) (and the formula is well-formed).

\[ \phi \Rightarrow \text{P}(\text{CALL p(a)} \supset Q) / (V, \text{env}, L) \]

\[ \phi \Rightarrow \text{P}(p \supset Q) / (V, \text{env}, L) \]

where \( a \in \text{FV} - (\text{FV}(\text{env} \cup \text{PROC}(L)) \cup \text{FV}(Q)) \).

\[ \phi \Rightarrow \exists z_1, \ldots, z_k \ . \ \text{P}(c \supset Q) / (V, \text{env}, L) \]

where \( L \cup L' = \emptyset, \ \text{VAR}(L') \cap V = \emptyset \) and \( \text{VAR}(L') = \{z_1, \ldots, z_k\} \).

\[ \phi \Rightarrow \text{P}(\text{CALL p(e)} \supset Q) / (V, \text{env}, L) \]

\[ \phi \Rightarrow \text{P}(\text{CALL p(e)} \supset \text{Q} \land V', / (V, \text{env}, L) \]

where \( V' \in \text{FV}(\text{env} \cup \text{PROC}(L)) \).

From the soundness and completeness results holding for the proof system \( \mathcal{R}^R \) (see the sections 5.3 and 5.4) we get that essentially the same results hold for the proof system \( \mathcal{T}^R \).

In the rest of this section we shall compare the proof system \( \mathcal{T}^R \) with various proof systems from the literature that can be used to prove total correctness of recursive procedure programs.

A proof system for total correctness of a language with recursive procedures is suggested by Sokolowski in /So77b/ and essentially the same rule can be found in /Ha79/ and /Ap81/.
\[ P(z) \langle \text{CALL} \ p \rangle Q \vdash P(z+1) \langle \text{CALL} \ p \rangle Q, \ \exists z P(z) \langle \text{CALL} \ p \rangle Q \]

where \( z \) is a new variable ranging over the natural numbers.

This rule turns out to be sufficient if we restrict our attention to programs with at most one recursively defined procedure. Harel proves in /Ha79/ that a version of the proof system formulated within the framework of dynamic logic is arithmetical sound and complete - the same result is proved in /Ap81/ within the framework of Hoare's logic.

Harel shows in /Ha79/ that (essentially) the rule /S/ above can be extended to cope with mutual recursive procedures. In the extended rule he only uses one counter although a number of procedures may be invoked. Intuitively, it is not necessary to have a counter for each procedure (as we have in \( J^R \)) because the procedures are declared at the same time and in a sense they behave as if there just was a single procedure (defined with a large case-statement).

In the case where procedure declarations are allowed to be nested it is, intuitively, not sufficient to have a single counter. When we enter the scope of a newly declared procedure we must be prepared to keep track of the number of recursive calls of it and when we leave its scope we do not bother further about it. Later in the proof we may meet the same procedure declaration again but in another context and then it has to be treated as a completely new procedure and as such it has nothing to do with the counter of the previous version of the procedure. This suggests that counters are closely related with the procedure declarations and this is reflected by the introduction of the component \( L \) of procedure name/(counter-) variable associations.
Recently, Meyer and Mitchell have given a proof system for total correctness properties of a language with nested recursive procedures /MeMi83/. Their idea is to extend the assertion language with some special formulas that can be used to express the effect of calling a given procedure. In this way the introduction of assumption lists is not needed: hypotheses about the recursive calls of the procedures can be included in the pre-conditions of the formulas. Furthermore, the introduction of counters is not needed any longer because of the additional power of the assertion language. However, Meyer and Mitchell's proof system is not a proof system in the usual style of Hoare's logic as presented in for instance /Ap81/, the reason being that the assertion language is not a first order language. So we can hardly compare our proof system $T^R$ with that of /MeMi83/.

We know of no Hoare-like proof system for proving total correctness of programs involving nested recursive procedures. However there are a few papers discussing partial correctness proof systems for such a language, for instance /Go75/, /ApdB77/, /Ap78/, /dB79/ and /C179/. It might be interesting to note that the proofs of soundness and completeness for these proof systems are rather complicated (see for instance /Ap78/) - just as those for the proof system $T^R$ given in the sections 5.3 and 5.4. However, in the proof of the completeness result in Section 5.4 even further complications arise because we prove total correctness properties (consider for instance the construction of the assumption lists).
The proof systems developed in the previous three chapters have been designed to prove quite general run-time properties, namely those that can be specified by time formulas. The worked examples of the three chapters show that, to a large extent, it is possible to obtain natural formalisations of the traditional informal analyses. However, there are certain details of the informal analyses that are not reflected directly in the formal proof. We have already mentioned the analysis of the (outer) loop of the bubble sorting algorithm in Section 3.5. Another aspect that is worth mentioning is that in the informal analyses we study a very special kind of run-time properties, namely the order of magnitude of the worst-case time complexity, whereas we in the formal proofs use the general time formulas to express run-time properties.

In order to narrow the gap between the informal analyses and the formal proofs we shall in this chapter consider a special purpose proof system for proving upper bounds on the run-time of programs. This proof system can be viewed as a step towards the one we really want for analysing the order of magnitude of the worst-case time complexity of programs. As in the previous chapters we shall be interested in the soundness and completeness properties of the proof system and based on worked examples we shall discuss the pragmatic issues. For the sake of simplicity we shall restrict our attention to the language of while programs.
In Section 6.1 we present the proof system \( U \) for proving upper bounds on the run-time of while programs. As an example of the use of the proof system we shall redo the analysis of the bubble sorting algorithm in Section 3.5 and we shall point out that we have obtained a more natural formalisation of the informal analysis. Finally, in Section 6.1 we prove that \( U \) can be derived from the previous proof system \( R \) (Chapter 3) and thereby that \( U \) is sound.

The question of completeness of the proof system \( U \) is more complicated and is discussed in the sections 6.2 and 6.3. In Section 6.2 we show that \( U \) is not complete in the same sense as \( R \), that is, it is not sufficient to assume that we have time formulas for the exact run-time requirements for each program. However, the result of Section 6.3 shows that we can impose conditions that will give us a completeness result but, unfortunately, these conditions are rather difficult to fulfill.

Section 6.4 contains a worked example. Using the proof system we can prove the \( \Theta(n^2) \) upper bound on Dijkstra's algorithm for finding the shortest paths in a graph. This analysis is based on the adjacency matrix representation of a graph. If the adjacency list representation is used, then an informal analysis using the so-called book-keeping trick gives an \( \Theta(e+n\log(n)) \) upper bound and we shall in Section 6.4 discuss to what extent this analysis can be formalised in \( U \). Finally, Section 6.5 contains some concluding remarks, among others, a discussion of how even more natural formalisations of the informal analyses can be obtained.

271
6.1 THE PROOF SYSTEM $\mathcal{U}$

We shall in this section present a proof system $\mathcal{U}$ for proving upper bounds on the run-time of while programs. The formulas of this proof system will have the form $P(c; T)Q/V$ where $P$, $c$, $Q$ and $V$ are as in the total correctness proof system $\mathcal{T}$ of Chapter 2 and $T$ is a so-called time term: it is a term of the assertion language with only program variables as free variables. The idea is, of course, that $T$ is a term for an upper bound on the run-time of $c$, so the formula $P(c; T)Q/V$ can be thought of as an abbreviation for the formula $P(c; \exists \text{time'}. \text{time} + \text{time}' = T)Q/V$ of the proof system $\mathcal{R}$ (where time' is a variable of sort nat, $\text{time}' \notin \text{FV}(T) \cup \{\text{time}\}$).

THE TIME-TERM EXPRESSIVENESS CONDITIONS

When we constructed the proof system $\mathcal{R}$ for run-time analysis of while programs in Chapter 3 we imposed a time expressiveness condition on the data type and its computational model in order to ensure that we have time formulas for the run-time requirements of the terms and boolean expressions of the language. Similarly, we shall now impose a condition that will ensure that we have time terms bounding the run-time requirements of the terms and boolean expressions.

Given a data type specified by a set $K$ of sorts and a $K$-sorted signature $\Sigma$ and given a computational model for it the time term expressiveness condition is fulfilled if

- for each sort $k$ of $K$ there is a term $U^+(x)$ with free variable $x$ of sort $k$ and such that for any state $s$

$$U^+(x)(s) \geq x(s)^+$$
- for each function symbol \( f \) of arity \((k_1, \ldots, k_m, k)\) there is a term 
  \[ U_f(x_1, \ldots, x_m) \]
  with free variables \( x_1, \ldots, x_m \) of sorts \( k_1, \ldots, k_m \),
  respectively, and such that for any state \( s \)
  
  \[ U_f(x_1(s), \ldots, x_m(s)) \geq f^+(x_1(s), \ldots, x_m(s)) \]
  
- for each relation symbol \( p \) of arity \( k_1, \ldots, k_m \) there is a term 
  \[ U_p(x_1, \ldots, x_m) \]
  with free variables \( x_1, \ldots, x_m \) of sorts \( k_1, \ldots, k_m \),
  respectively, and such that for any state \( s \)
  
  \[ U_p(x_1(s), \ldots, x_m(s)) \geq p^+(x_1(s), \ldots, x_m(s)) \].

Note that if the exact time term expressiveness condition introduced
in Section 3.6 is fulfilled then so is the time term expressiveness
condition defined above.

**Example 6.1-1:** It is straightforward to check that the following data
types and computational models satisfy the time term expressiveness
condition:

- the data type of Peano Arithmetic and its computational model
  (Example 2.1-1 and 3.1-1),

- the data type of Extended Peano Arithmetic and its computational
  model (Example 3.1-2),

- the data type of one-dimensional arrays and its computational
  model (Example 3.1-3).

If the time term expressiveness condition is fulfilled we will
have time terms \( U(e) \) and \( U(b) \) bounding the run-time required to
evaluate the term \( e \) and the boolean expression \( b \), respectively. The
time terms are defined by structural induction on the term/boolean
expression exactly as in Section 3.6 so we omit the details here.
Corresponding to the lemmas 3.2-1 and 3.6-1 we have
Lemma 6.1 - If the time term expressiveness condition holds for the data type and its computational model then for every term \( e \), boolean expression \( b \) and state \( s \)

\[
- U(e)(s) \geq e^S(s), \\
- U^S(e)(s) \geq e^S(s) + e(s)^+, \\
- U(b)(s) \geq b^S(s).
\]

Here \( U^S(e) \) is an abbreviation for \( U(e) + U^+(e) \).

The proof system

As mentioned the formulas of the proof system \( \mathcal{U} \) have the form

\[ P(c:T)Q/V \]

where \( P, c, Q \) and \( V \) are as in the proof system \( \mathcal{T} \) (Chapter 2) and where \( T \) is a time term. We say that \( P(c:T)Q/V \) is a well-formed formula of \( \mathcal{U} \) if

\[
- FV(c) \subseteq V, \\
- FV(c) \subseteq V, FV(Q) \subseteq V \text{ and } FV(T) \subseteq V.
\]

The validity of the formula, written \( \mathcal{F}P(c:T)Q/V \), is now defined to mean that

for every state \( s \) satisfying \( \mathcal{F}P(s) \) there is a state \( s' \) and a natural number \( r \) such that

\[ \langle c, s \rangle \rightarrow s', \mathcal{F}Q(s, s') \text{ and } r \in T(s). \]

Note that if \( P(c:T)Q/V \) is a well-formed formula of \( \mathcal{U} \) then the formula \( P(c: \exists \text{time'}. \text{time}+\text{time'}=T)Q/V \) is well-formed (for \( \mathcal{R} \)). Furthermore, if \( \mathcal{F}P(c:T)Q/V \) holds then so does \( P(c: \exists \text{time'}. \text{time}+\text{time'}=T)Q/V \) and vice versa.

To simplify the notation used in the axioms and rules below we
shall, given two time terms $T_1$ and $T_2$, write $T_1 T_2$ as an abbreviation for the formula $\exists \text{time}'. T_1 + \text{time}' = T_2$ ($\text{time}'$ is a variable of sort $\text{nat}$ satisfying $\text{time}' \notin \text{FV}(T_1) \cup \text{FV}(T_2)$).

Assuming that the time term expressiveness condition is fulfilled (that is, we have the time terms $U(e)$, $U^S(e)$ and $U(b)$) we have the following axioms and rules in the proof system $\mathcal{U}$.

The proof system $\mathcal{U}$

<table>
<thead>
<tr>
<th>Assumption</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle x : e : U^S(e) \rangle I_{V} {x } \wedge x = e / V$</td>
<td>/ass-$\mathcal{U}$</td>
</tr>
<tr>
<td>$P \wedge b \langle c_1 : T \rangle Q / V, P \wedge b \langle c_2 : T \rangle Q / V$</td>
<td>/IF-$\mathcal{U}$</td>
</tr>
<tr>
<td>$P (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 : U(b) + T') Q / V$</td>
<td></td>
</tr>
<tr>
<td>$P \langle c_1 : T \rangle P' \langle Q_1 / V, P' \langle c_2 : T \rangle Q_2 / V, Q_1 \rightarrow T_2 T_2'$</td>
<td>/;-$\mathcal{U}$</td>
</tr>
<tr>
<td>$P \langle c_1 : T + T_1 T_2 \rangle Q_1 \cdot Q_2 / V$</td>
<td></td>
</tr>
<tr>
<td>$P(z + 1) \wedge b \langle c : T' \rangle P(z) Q' / V \cup {z}, P(0) \rightarrow \neg b, P(z) \wedge b A \rightarrow Q, Q' \rightarrow Q, P(z) \rightarrow Q$</td>
<td>/WHILE-$\mathcal{U}$</td>
</tr>
<tr>
<td>$P(z) \wedge b \rightarrow U(b) \times T', Q' \rightarrow T_4 T_4, U(b) + T' + T_4 T_4$</td>
<td></td>
</tr>
<tr>
<td>$\exists z. P(z) \langle \text{WHILE } b \text{ DO } c : T \rangle Q / V$</td>
<td></td>
</tr>
<tr>
<td>where $z$ is a variable of sort $\text{nat}$ satisfying $z \notin V$</td>
<td></td>
</tr>
<tr>
<td>$P \rightarrow P'$, $P' \langle c : T' \rangle Q' / V, Q' \rightarrow Q, P \rightarrow T' + T$</td>
<td>/cons-$\mathcal{U}$</td>
</tr>
<tr>
<td>$P \langle c : T \rangle Q / V$</td>
<td></td>
</tr>
<tr>
<td>$P \langle c : T \rangle P A Q / V$</td>
<td>/inv-$\mathcal{U}$</td>
</tr>
</tbody>
</table>

We shall write $\mathcal{U} + P(c : T \rangle Q / V$ if the formula $P(c : T \rangle Q / V$ can be proved using the axioms and rules above (and with the constraint that all formulas from the assertion language that are used in /;-$\mathcal{U}$, /WHILE-$\mathcal{U}$,

275
and $\text{/cons-}\mathcal{U}$ must be true and all the formulas of $\mathcal{U}$ must be well-formed).

The axioms and rules of $\mathcal{U}$ are derived from those of $\mathcal{R}$ by restricting the time formulas to have the very special form $\text{time}(T)$. Assuming that the formulas of the hypotheses of the rules of $\mathcal{R}$ have this form it need not be the case that those of the conclusions have that form. Therefore we have added some extra assumptions to the rules that will bring the time formulas back to the right form.

**Example**

As an example of the use of the proof system $\mathcal{U}$ we shall consider the bubble sorting algorithm of Section 3.5 once more. To a large extent a proof of the formula

$$ \text{TRUE(bubble sorting:32 \text{length}(l) \text{length}(l)+5)TRUE/V} $$

in $\mathcal{U}$ will be a direct reformulation of that in $\mathcal{R}$ as presented in Section 3.5. Nonetheless, there are some important differences that bring the proof in $\mathcal{U}$ closer to the informal analysis of the algorithm (which we also considered in Section 3.5). We shall therefore redo a part of the proof in $\mathcal{U}$.

Remember that the bubble sorting algorithm basically consists of two nested loops. For the body, called outer, of the outer loop we can (very much as in Section 3.5) construct a proof of the formula

$$ P(z+1) \land i = 0 \langle \text{outer:32'(i-1)+9} \rangle P(z) \land i \not\in V \cup \{z\} $$

in $\mathcal{U}$ (where $P(z)$ is the formula $i(z)$. The next step will then be to apply the while rule $\text{/WHILE-}\mathcal{U}$ but in order to do so we have to find time terms $T$ and $\tilde{T}$ such that
hold \( U(n=0) \) can be chosen to be the term 3. A solution is here to choose \( T \) to be \( 32'i+i+3 \) and \( \overline{T} \) to be \( 32'i'(i-1)-9 \) so we get a proof of

\[ \exists z. P(z) \langle \text{WHILE } n=0 \text{ DO outer: } 32'i'i+3 \rangle \text{TRUE/V.} \]

The rest of the proof is now essentially as in Section 3.5 and is therefore omitted.

The conditions (1), (2) and (3) above should be compared with those obtained in the proof in \( \mathcal{R} \) where we have to find a time formula \( R \) such that

\[ \forall P(z) \wedge i=0 \wedge \text{time=3} \rightarrow R \]

and

\[ \forall (\text{time}=3) \Theta (\text{time} 32'i'(i-1)+3) \Theta (i<\overline{i})' R)) \rightarrow R \]

hold. The conditions of \( \mathcal{U} \) look more like an ordinary recurrence relation than those of \( \mathcal{R} \) and it should therefore be easier to apply well-known techniques to solve them. As we already mentioned in Section 3.5 we can obtain a solution to (1), (2) and (3) by, essentially, using the technique of summing series and we are then very close to the informal analysis where we calculate the sum

\[ \Theta (\sum_{i=1}^{\text{length}(1)} (i-1)). \]

\( \mathcal{R} \text{ IS AS POWERFUL AS } \mathcal{U} \)

To further discuss the connections between the proof systems \( \mathcal{R} \) and \( \mathcal{U} \) we shall now show that anything provable in \( \mathcal{U} \) is also provable
Lemma 6.1-2: If the time term expressiveness condition as well as the
time expressiveness condition is fulfilled for the data type and
its computational model then for every well-formed formula
\( P\langle c: T \rangle Q/V \) of \( U \)
\[ U \vdash P\langle c: T \rangle Q/V \text{ implies } \mathcal{R} \vdash c: \text{time} T \rangle Q/V. \]

Proof: First note that the time term expressiveness condition together
with the time expressiveness condition give that \( \mathcal{P} \mathcal{E}^S(e) \rightarrow \text{time} U^S(e) \)
and \( \mathcal{P} \mathcal{E}(b) \rightarrow \text{time} U(b) \) hold (see the lemmas 3.2-1 and 6.1-1) The proof
of the lemma is by induction on the length of the proof in \( U \). The
cases where the last axiom or rule applied is one of \( \text{ass-} U/ \), \( \text{IF-} U/ \),
\( \text{cons-} U/ \) and \( \text{inv-} U/ \) are straightforward and therefore omitted. The
cases of \( /; -U/ \) and \( \text{WHILE-} U/ \) are as follows:

Case \( /; -U/ \): Assume now that we from the proofs

1. \[ U \vdash P\langle c_1 : T_1 \rangle P' \wedge Q_1 /V, \]
2. \[ U \vdash P' \langle c_2 : T_2 \rangle Q_2 /V \]
and
3. \[ \mathcal{P} Q_1 \rightarrow T_2 \langle \rho \rangle \]
get the proof
\[ U \vdash P\langle c_1 : T_1 + T_2 \rangle Q_1 . Q_2 /V. \]
The induction hypothesis can be applied to the proofs (1) and (2) and
gives the proofs
\[ \mathcal{R} \vdash P\langle c_1 : \text{time} T_1 \rangle P' \wedge Q_1 /V \]
and

278
\[ R \vdash P'(c_2; \text{time} \leq T_2) Q_2 / V. \]

Using /;R we then get the proof
\[ R \vdash P(c_1; c_2; (\text{time} \leq T_1) \land (Q_1 \land (\text{time} \leq T_2))) Q_1 Q_2 / V. \]

From (3) we get that
\[ q_1 (\text{time} \leq T_2) \rightarrow \text{time} \leq T_2 \]
so
\[ (\text{time} \leq T_1) \land (q_1 (\text{time} \leq T_2)) \rightarrow \text{time} \leq T_1 + T_2 \]
and then /cons-R/ gives
\[ R \vdash c_1; c_2; \text{time} \leq T_1 + T_2 Q_1 Q_2 / V \]
as required.

Case /WHILE-U/: Assume now that we from

1. \[ \forall z + 1 \lambda b \langle c : T' \rangle P(z) Q' / V \forall (z), \]
2. \[ P(0) \rightarrow \neg b, \ P(z) \land b \rightarrow Q, \ \neg P', Q \rightarrow Q, \]
3. \[ \neg Q' \rightarrow T \neg T, \]
4. \[ \neg U(b) + T' \neg T \]

we have a proof
\[ \forall z. P(z) (\text{WHILE } b \ DO \ c : T) Q / V. \]

The induction hypothesis applied to (1) gives
\[ R \vdash P(z + 1) \wedge \langle c : \text{time} \leq T' \rangle P(z) Q' / V \forall (z). \]

From (2) (and \( E(b) \rightarrow \text{time} \leq U(b) \)) we get
\[ P(z) \land b \rightarrow \neg E(b) \rightarrow \text{time} \leq T. \]

From (3) we have
so using (4) and that \( \forall E(b) \to_{\text{time}} U(b) \) we get

\[ \forall E(b) \Phi_{\text{time}}(\forall Q'(\text{time} T')) \to_{\text{time}} T. \]

Now using /WHILE-\( R \)/ we get the required proof of

\[ \forall z. P(z)(\text{WHILE } b \text{ DO } c_{\text{time} T} Q/V). \]

Using this result together with The Soundness Theorem for \( R \) (Section 3.3) we get that the proof system \( U \) is sound:

The Soundness Theorem for \( U \)

Given a data type and a numerical computational model for it, if the time term expressiveness condition as well as the time expressiveness condition are fulfilled then for every well-formed formula \( P_{\langle c:T \rangle Q/V} \) of \( U \)

\[ \forall P_{\langle c:T \rangle Q/V} \text{ implies } \exists P_{\langle c:T \rangle Q/V}. \]

A straightforward and direct proof of this result shows that it is not necessary that the time expressiveness condition holds in order for the soundness result to hold. We omit the details.

6.2 A NEGATIVE Completeness RESULT FOR \( U \)

We have now seen that the proof system \( R \) of Chapter 3 is as powerful as \( U \). Of course we cannot expect \( U \) to be as powerful as \( R \) for the simple reason that \( R \) allows one to prove arbitrary time formulas holding for the run-time of a program whereas \( U \) is restricted to time formulas expressing upper bounds on the run-time. The question we can ask is therefore whether results about upper bounds on the run-time that can be proved in \( R \) also can be proved in \( U \). Or, put in
another way, is the proof system $\mathcal{U}$ complete in the same sense as $\mathcal{R}$? As we shall prove in this section the answer is no!

Given a data type and a numerical computational model for it, we shall say that the expressiveness condition for $\mathcal{U}$ is fulfilled if

- the time term expressiveness condition is fulfilled
- the expressiveness condition for $\mathcal{R}$ is fulfilled (see Section 3.4).

The last condition ensures that we have relational formulas $G[c]$ and time formulas $E[c]$ expressing the graph and the exact run-time requirements, respectively, for each while program $c$.

**Example 6.2-1**: The data type of Extended Peano Arithmetic and its uniform computational model (Example 3.1-2) satisfy the expressiveness condition for $\mathcal{U}$. From Example 6.1-1 we get that the time term expressiveness condition is fulfilled and from Example 3.4-1 we get that the expressiveness condition for $\mathcal{R}$ is fulfilled as well. \\

THE NEGATIVE COMPLETENESS RESULT AND ITS PROOF

Using this expressiveness concept we shall prove the following result:

The Negative Completeness Theorem for $\mathcal{U}$

There exists a data type with a numerical computational model such that the expressiveness condition for $\mathcal{U}$ is fulfilled but for some well-formed formula $P(c;T)Q/V$ of $\mathcal{U}$ we have

$\vdash P(c;T)Q/V$ but not $\mathcal{U}\vdash P(c;T)Q/V$. \\

The proof will be by contradiction. We shall consider the data type of Extended Peano Arithmetic and its uniform computational
model (see Example 3.1-2). From Example 6.2-1 we have that the expressiveness condition for $\mathcal{U}$ is fulfilled. We shall now prove the following lemma:

**Lemma 6.2-1**: Given the data type of Extended Peano Arithmetic and its uniform computational model there exists a program $\mathcal{C}$ and a program variable $y$, $y \notin \text{FV}(\mathcal{C})$, such that for some constants $k$ and $k'$

$$\mathcal{C} \vdash \text{TRUE} \leq_{y+k'} \text{TRUE}/\text{FV}(\mathcal{C})$$

but the formula is not provable in $\mathcal{U}$.

The program $\mathcal{C}$ will have the form $c; c_0; c_1$ where

$$c_0 \equiv \text{IF } z=0 \text{ THEN } c \text{ ELSE } z:=z$$

and

$$c_1 \equiv \text{IF } z=1 \text{ THEN } c \text{ ELSE } z:=z.$$ 

The program $c$ will be constructed below. For the moment it is sufficient to know that $c$ operates as follows on a state $s$:

A) the value $0$ is assigned to all the variables of $\text{FV}(c)$ except $x$ and $y$,

B) the value of $z$ (which is either $0$ or $1$) is computed in such a way that the values of $x$ and $y$ are not changed and at most $y(s)+k$ time units are used,

C) if less than $y(s)+k$ time units have been used so far then the remaining time is spent in a "do nothing" loop such that the exact execution time becomes $y(s)+k$.

Here $k$ is a constant independent of the input to the program.

From B and C it follows that $c$ terminates on any input. Since
the values of x and y are not changed by c (confer B) and since the values of all other variables do not matter (confer A) we see that exactly the same computation will be performed each time c is executed in the program c;c_0;c_1 and, especially, it will result in the same final value of z. This value will be either 0 or 1 (confer B) so c is executed exactly two times in c;c_0;c_1. This is illustrated on the following figure:

Therefore the exact run-time of the program c;c_0;c_1 in the state s is 2'(y(s)+k)+8 (three time units are required for the tests z=0 and z=1, respectively, and the assignment z:=z requires two time units). This proves that the formula

TRUE(c;c_0;c_1:2'(y+k)+8)TRUE/FV(c)

is a valid formula and then by assumption it is provable in the proof system U:

(1) U↑TRUE(c;c_0;c_1:2'(y+k)+8)TRUE/FV(c).

We shall now investigate how this proof might have been obtained from proofs of formulas for the subprograms c, c_0 and c_1. We have the following result

**Lemma 6.2-2:** If U↑P(c_1;c_2:T)Q/V then there are formulas P', P'', Q_1 and Q_2 and time terms T_1, T_2 and T_2 such that

- U↑P→P', U↑P→T_1+T_2∧T and U↑P∧Q_1→Q,
- U↑P''<c_1:T_1>P''∧Q_1/V, U↑P''<c_2:T_2>Q_2/V and U↑P''→T_2∧T_2'.

///

283
The proof of this result is straightforward by induction on the length of the proof of $P(c_1; c_2; T)Q/V$ in $U$. The last rule applied in the proof must be one of $/; - U$, $/ \text{cons} - U$ and $/ \text{inv} - U$ and each of these cases are straightforward. We omit the details.

The proof of (1) has been obtained either by viewing the program $c;c_0;c_1$ as $(c;c_0);c_1$ or as $c;(c_0;c_1)$. The two cases are quite similar so let us consider the first. Here Lemma 6.2-2 gives that we have formulas $P', P_i$, $Q'$ and $Q_1$ and time terms $T'$, $T_1$ and $\overline{T_1}$ such that

$$\begin{align*}
& (2) \quad \not{\text{TRUE}} \rightarrow P', \not{\text{TRUE}} \rightarrow T' + \overline{T_1} < 2^*(y+k) + 8 \text{ and } \not{\text{TRUE}} \rightarrow Q', Q_1 \rightarrow \text{TRUE}, \\
& (3) \quad U \rightarrow P' \not{\langle c; c_0 \rangle} \not{\rightarrow} P_1 \sim Q'/FV(c), \\
& (4) \quad U \rightarrow P_1 \not{\langle c_1 \rangle} \not{\rightarrow} Q_1/FV(c) \\
\text{and} \\
& (5) \quad \not{Q'} \rightarrow T_1 \not{\overline{T_1}}.
\end{align*}$$

Applying Lemma 6.2-2 once more but to the proof of (3) we get that there are formulas $P$, $P_0$, $Q$ and $Q_0$ and time terms $T$, $T_0$ and $\overline{T_0}$ such that

$$\begin{align*}
& (6) \quad \not{P} \rightarrow P, \not{P'} \rightarrow T + \overline{T_0} \not{\rightarrow} T' \text{ and } \not{\text{P'}} \rightarrow Q', Q_0 \rightarrow P_1 \sim Q', \\
& (7) \quad U \rightarrow P \not{\langle c \rangle} \not{\rightarrow} P_0 \sim Q/FV(c), \\
& (8) \quad U \rightarrow P_0 \not{\langle c_0 \rangle} \not{\rightarrow} Q_0/FV(c) \\
\text{and} \\
& (9) \quad \not{Q} \rightarrow T_0 \not{\overline{T_0}}.
\end{align*}$$

The Soundness Theorem for $U$ gives that the formulas (7), (8) and (4) are valid.

We shall now see that the terms $T$, $T_0$ and $T_1$ must be time terms for the exact run-time of $c$, $c_0$ and $c_1$, respectively. Consider some
state $s$. From (2) and (6) we get that $kP(s)$ holds and the validity of (7) gives that for some $s'$ and $r$

$$\langle c, s\rangle \rightarrow s', kP_0(s', s') \text{ and } r \notin T(s).$$

Since $kP_0(s')$ holds we get from the validity of (8) that for some $s''$ and $r_0$

$$\langle c_0, s'\rangle \rightarrow r_0 s'', kP_0(s', s'') \text{ and } r_0T_0(s').$$

From $kQ(s, s'), kQ_0(s', s''), (2)$ and (6) we get that $kP_1(s'')$ holds.

So from the validity of (4) we get that for some $s'''$ and $r_1$

$$\langle c_1, s''\rangle \rightarrow r_1 s''', kP_1(s'', s''') \text{ and } r_1T_1(s'').$$

Since $kQ(s, s')$ we get from (9) that $T_0(s') \leq T_0(s)$. Similarly, from $kQ(s, s'), kQ_0(s', s'')$, (2), (6) and (5) we get $T_1(s'') \leq T_1(s)$ so we have

$$r + r_0 + r_1 \leq (T + T_0 + T_1)(s).$$

From (2) and (6) we get that

$$(T + T_0 + T_1)(s) \leq 2'(y(s) + k) + 8.$$ 

However, we know from earlier that the exact run-time of $c; c_0; c_1$ from the state $s$, that is $r + r_0 + r_1$, is equal to $2'(y(s) + k) + 8$. So it must be the case that $r = T(s)$, $r_0 = (T_0(s') = T_0(s'))$ and $r_1 = (T_1(s') = T_1(s''))$.

This proves that the terms $T$, $T_0$ and $T_1$ are terms for the exact run-time of the programs $c$, $c_0$ and $c_1$, respectively.

The program $c$ will be constructed such that the following additional condition is fulfilled

D) there are two different sets of input to $c$ such that on the one set $z$ gets the value 0 and on the other set the value 1.

If the state $s$ considered above is such that $z(s') = 0$ then we will
have \( r_0 = y(s) + k + 3 \) and thereby \( T_0(s) = y(s) + k + 3 \). On the other hand, if \( z(s') = 1 \) then \( T_0(s) = 5 \). The idea is now to make the computation of the value of \( z \) in \( c \) so complicated that the exact run-time of \( c_0 \) (and \( c_1 \)) cannot be expressed as a term. This will give us the required contradiction and thereby prove the negative completeness result.

We shall now describe the computation performed by the program \( c \) in more detail. The terms of the data type of Extended Peano Arithmetic can be coded as natural numbers (there are a countable number of constant symbols, a countable number of variables and a finite number of function symbols). This coding can be chosen such that every natural number \( n \) can be viewed as a coding of some term which we shall denote \( \text{encode}(n) \) (note the encoding need not be unique).

We shall now define a function \( \text{eval} \) with two parameters \( n \) and \( m \). It will evaluate the term \( \text{encode}(n) \) in the state \( s \) with \( x(s) = n \), \( y(s) = m \) and \( x'(s) = 0 \) for all other variables \( x' \). It is straightforward to verify that \( \text{eval} \) is a primitive recursive function (see for example /Sh67/). The language of while programs with the data type of Extended Peano Arithmetic is powerful enough to compute any partial recursive function. This means that there is a program \( \text{eval} \) computing the function \( \text{eval} \). Without loss of generality we can assume that the following property holds:

for any pair \( (s, s') \) of states and natural number \( r \)

\[ \langle \text{eval}, s \rangle \xrightarrow{r} s' \]

if and only if

\[ z(s') = \text{eval}(x(s), y(s)), x(s') = x(s) \text{ and } y(s') = y(s). \]

Logically, the program \( \text{eval} \) performs two main tasks. First it encodes the value of the variable \( x \) in order to determine the term.
This encoding can be done within a run-time that only depends on the initial value of $x$; especially, it does not depend on the value of $y$. The second task is to evaluate the term. The run-time of this process depends only on the structure of the term and not on the state in which we evaluate it - remember, we consider the uniform computational model of the Extended Peano Arithmetic. So especially, the run-time of the second task does not depend on the value of $y$:

$$
\begin{cases}
\text{for any three states } s, s' \text{ and } s'' \text{ and for any three natural numbers } m, r \text{ and } r',
\text{if } \langle \text{eval}, s_r \rangle \rightarrow s' \text{ and } \langle \text{eval}, s_y^m \rangle \rightarrow s'' \text{, then } r = r'.
\end{cases}
$$

Note the discussion above makes it only plausible that the program eval can be constructed such that (§) holds. A detailed proof will involve the construction of the program. We shall refrain from that.

In order to construct the program $c$ we shall insert statements in the program eval that will count the run-time used so far in a special variable clock ($\text{clock}^FV(\text{eval})$) and stop the execution of eval when the value of clock exceeds that of $y$. This transformation of eval is defined structurally as in the table below. We have

$$
\begin{cases}
\text{for any state } s \text{ there exists natural numbers } n, r \text{ and } r',
\text{and a state } s' \text{ such that }
\langle \text{eval}, s_y^n \rangle \rightarrow s',
\text{if and only if }
\langle \text{eval}, s_y^0 \text{ clock} \rangle \rightarrow s'_y \text{ clock and } n > r',
\end{cases}
$$

This means that for sufficient large values of $y$ eval behaves in the same way as eval.
If initially, the value of the variable `clock` exceeds that of `y` then the execution of `eval` will require a constant amount of time, $k_1$, only determined by the structure of `eval`. If the value of `clock` exceeds that of `y` some time during the execution of `eval` then we can bound the difference between the final value of `clock` and that of `y` by a constant, $k_2$, that only depends on the structure of `eval`. The following table defines the constants $k_1(c)$ and $k_2(c)$ for any program $c$:

<table>
<thead>
<tr>
<th>$c$</th>
<th>$k_2(c)$</th>
<th>$k_2(c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x:=e$</td>
<td>7</td>
<td>$U^S(e)+8$</td>
</tr>
<tr>
<td>IF $b$ THEN $c_1$</td>
<td>7</td>
<td>$\max{U(b)+8+k_1(c_1), k_2(c_1)}$</td>
</tr>
<tr>
<td>ELSE $c_2$</td>
<td></td>
<td>$U(b)+8+k_1(c_2), k_2(c_2)$</td>
</tr>
<tr>
<td>$c_1;c_2$</td>
<td>$k_1(c_1)+7$</td>
<td>$\max{k_2(c_1)+7, 6+k_1(c_2), k_2(c_2)}$</td>
</tr>
<tr>
<td>WHILE $b$ DO $c$</td>
<td>$U(b)+9$</td>
<td>$\max{U(b)+8+k_1(c), k_2(c)+U(b)+9}$</td>
</tr>
</tbody>
</table>

The program $c$ can now be described as follows:
- initialise all the variables of the set \((FV(eval) - \{x, y\}) \cup \{\text{clock}\}\) to zero,

- execute the statement \(\text{clock} := n\) where \(n\) is two times the number of variables in the set \((FV(eval) - \{x, y\}) \cup \{\text{clock}\}\) plus two,

- execute the program \(eval\),

- execute the following piece of program

  \[
  \text{IF } z = 0 \text{ THEN } z := 0 \text{ ELSE } z := 1; \text{ clock} := \text{clock} + 9,
  \]

- execute a program that takes time \((y + k) - \text{clock}\) where \(k\) is the constant \(k_2(\text{eval}) + n + 65\).

The last subprogram can be written as a while loop where most of the time is spent, followed by a number of nested conditionals that takes care of the remaining "few" time units. The overhead of 65 time units is due to that part of the program (and are, of course, subject to how it is actually coded).

It is straightforward to check that the assumptions A, B, C and D imposed earlier are indeed fulfilled.

The proof of the negative completeness result can now be completed by a diagonalisation argument. Let \(e\) be the term

\[
\frac{(\alpha_0^2 - 5)}{(y + k - 2)}
\]

and let \(n\) be a coding of \(e\). Let \(s\) be the state with \(x(s) = n\), \(y(s) = m\) and \(x'(s) = 0\) for \(x' \not\in \{x, y\}\). Here \(m\) is a natural number that is large enough to ensure that \(eval\) behaves as \(eval\) (confer the property (\$)). This means that if \(\langle c', s \rangle \rightarrow_s s'\) then

- \(z(s') = 0\) if and only if \(e(s) = 0\)
- \(z(s') = 1\) if and only if \(e(s) > 0\)

Consider now the program \(c_0\). From earlier we have
\[ z(s') = 0 \text{ implies that } \tilde{T}_0(s) = y(s) + k + 3. \]

\[ z(s') = 1 \text{ implies that } \tilde{T}_0(s) = 5. \]

We now see that if \( e \) and \( s \) are such that \( e(s) = 0 \) then \( z(s') = 0 \) and thereby \( \tilde{T}_0(s) = y(s) + k + 3. \) But then \( e(s) = 1 \) follows from the definition of \( e. \) On the other hand, if \( e(s) > 0 \) then \( z(s') = 1 \) and thus \( \tilde{T}_0(s) = 5. \) But then \( e(s) = 0 \) follows from the definition of \( e. \) This gives the required contradiction and we have proved Lemma 6.2-2 and thereby the negative completeness result for \( U. \)

### 6.3 A WEAK COMPLETENESS RESULT FOR \( U \)

Intuitively, the reason for why \( U \) is not complete in the same sense as \( A \) (see Section 6.2) is that the assertion language contains too few terms. One could therefore suggest strengthening the expressiveness condition so we have time terms for the exact runtime requirements of every while program. As we shall see in this section this is indeed sufficient to give a completeness result for \( U \) but unfortunately the assumption is really too strong.

Given a data type and a numerical computational model for it, we say that the strong expressiveness condition for \( U \) is fulfilled if

- the exact time term expressiveness condition is fulfilled,
- for every while program \( c \) there is a relational formula \( G[c] \) with \( \text{FV}(G[c]) \subseteq \text{FV}(c) \cup \text{FV}(c) \) and satisfying that for every pair \( (s, s') \) of states
  \[ \models G[c](s, s') \]
  if and only if
  \[ (c, s) \stackrel{r}{\rightarrow} s' \text{ for some state } s' \text{ and natural number } r \text{ and } s'' = \text{FV}(c)^r s', \]
for every while program $c$ there is a time term $U[c]$ with $FV(U[c])\subseteq FV(c)$ and satisfying that for every pair $(s,r)$ of state and natural number
\[ \langle c, s \rangle \rightarrow^* \text{ s', for some state s'} \]
implies
\[ U[c](s) = r. \]

The last condition thus says that if the program $c$ terminates when executed from the state $s$ then $U[c](s)$ will be the exact run-time of the computation. We shall discuss the fulfillment of this condition later.

Note that if the strong expressiveness condition for $U$ is fulfilled then so is the expressiveness condition for $U$: given a program $c$ we can define $E[c]$ to be the formula $(\exists X'. G[c]_{X'X}^X)^{\text{time}=U[c]}$ (where $X$ is a vector over the variables of $FV(c)$ and $X$ and $X'$ are as usual).

THE COMPLETENESS RESULT FOR $U$ AND ITS PROOF

We shall in the following restrict ourselves to the data types with minus operators and their numerical computational models with subtraction. A data type with a minus operator is a data type whose signature contains the function symbol $-$ of arity $(\text{nat nat,nat})$. A numerical computational model with subtraction is a numerical computational model where the symbol $-$ is interpreted as subtraction of natural numbers. Then we have
The Completeness Theorem for \( \mathcal{U} \)

Given a data type with a minus operator and given a numerical computational model with subtraction for it, if the strong expressiveness condition for \( \mathcal{U} \) is fulfilled then for every well-formed formula \( P(c:T)Q/V \) of \( \mathcal{U} \)

\[
P(c:T)Q/V \text{ implies } \mathcal{U}P(c:T)Q/V.
\]

The proof of this result is by structural induction on the program \( c \). The proof is an extension of that proving the completeness result for the proof system \( \mathcal{T} \) for total correctness in Section 2.4.

Case \( x:=e \): Assume now that

\[
P(x:=e:T)Q/V
\]

and we shall construct a proof of the formula in \( \mathcal{U} \). From /ass-\( \mathcal{U} \) we get a proof of

\[
P(x:=U^S(e)\rightarrow V)Q/V.
\]

As in the corresponding case in the completeness proof for \( \mathcal{T} \) we have \( \mathcal{U}(x:=\neg x \rightarrow Q) \) and below we shall prove that

\[
P \rightarrow U^S(e)\mathcal{T}.
\]

So using first /inv-\( \mathcal{U} \) and then /cons-\( \mathcal{U} \) we get the required proof of the formula of (1).

To prove (2) assume that \( \mathcal{U}P(s) \) holds for some state \( s \). From (1) we get that for some \( s' \) and \( r \)

\[
<x:=e,s>s', PQ(s,s') \text{ and } \mathcal{T}(s).
\]

From /ass-\( \mathcal{U} \) and Lemma 3.1-3 we get that \( r = e^S(s) + e(s)^+ \). But from Lemma 6.1-1 we get \( U^S(\emptyset(s)) = e^S(s) + e(s)^+ \) and thereby \( \mathcal{U}(U^S(e)\mathcal{T})(s) \).

292
Case IF b THEN \(c_1\) ELSE \(c_2\): Assume that

(1) \(\not\forall P\langle IF \ b \ THEN \ c_1 \ ELSE \ c_2 : T \rangle Q/V\)

and we shall construct a proof of the formula in \(\mathcal{U}\). Define \(T'\) to be the time term \(T-U(b)\). Below we shall prove that the formula

(2) \(\not\forall P\land b\langle c_1 : T' \rangle Q/V\)

is valid and similarly it can be proved that the formula

\(\not\forall P\land b\langle c_2 : T' \rangle Q/V\)

is valid. The induction hypothesis can then be applied and gives us proofs of the two formulas in \(\mathcal{U}\). Using /IF-U/ we then get a proof of

\(\not\forall P\langle IF \ b \ THEN \ c_1 \ ELSE \ c_2 : U(b)+T' \rangle Q/V\).

Since \(\not\forall P\rightarrow U(b)+T' \leq T\) clearly holds we can apply /cons-U/ and get a proof of the formula of (1) as required.

To prove that (2) is valid assume that \(\not\forall P\land b(s)\) holds for some state \(s\). From (1) we get that for some \(s'\) and \(r\)

\(\langle IF \ b \ THEN \ c_1 \ ELSE \ c_2 : s \rangle \rightarrow s', \forall Q(s,s')\) and \(r \in T(s)\).

Then /IF-\$\$/ gives that

\(\langle c_1 ,s \rangle \rightarrow s'\)

where \(r=b^S(s)+r'\). From Lemma 6.11 we have \(U(b)(s)=b^S(s)\) so \(r' \in (T-U(b))(s)\) holds and thereby the validity of (2) has been proved.

Case \(c_1 ; c_2\): Assume that

(1) \(\not\forall P\langle c_1 ; c_2 : T \rangle Q/V\)

holds. We shall construct a proof of the formula in \(\mathcal{U}\). As in the
completeness proof for $T$, case $c_1; c_2$, we define $Q_1$, $Q_2$ and $P'$ to be the formulas $G[c_1][I_{-FV}(c_1)]$, $G[c_2][I_{-FV}(c_2)]$ and $\exists x'. G[c_2][X'X]$ (where $X$, $\bar{X}$ and $X'$ are "as usual"), respectively. Furthermore, we define the time terms $T_1$, $T_2$ and $\hat{T}_2$ to be $U[c_1]$, $U[c_2]$ and $T-U[c_1]$, respectively. A straightforward modification of the proof for the similar results in the completeness proof for $T$, case $c_1; c_2$, shows that the formulas

$$P(c_1;T_1)P'\wedge Q_1/V$$

and

$$P'(c_2;T_2)Q_2/V$$

are valid. The induction hypothesis then gives that they are provable in $U$. Using $/inv-U/$ we therefore get a proof of the formula

$$P(c_1;T_1)P'\wedge \overline{PAQ_1}/V.$$ 

Below we shall prove that

$$(2) \overline{PAQ_1} \rightarrow T_2 \hat{T}_2$$

so using $/\,-U/$ we get a proof of

$$P(c_1;c_2;T_1+\hat{T}_2)(\overline{PAQ_1})Q_2/V.$$ 

As in the completeness proof for $T$, case $c_1; c_2$, we have

$$\overline{PAQ_1}Q_2 \rightarrow Q$$

and it is easy to verify that $\overline{P} \rightarrow T_1+\hat{T}_2$. So $/cons-U/$ gives us the required proof of the formula of (1).

To prove (2) assume that $\overline{PAQ_1}(s,s')$ for some pair $(s,s')$ of states. From $\overline{P}(s)$ and (1) we get that for some $r$ and $s''$

$$<c_1; c_2,s> \not\in s'', \overline{Q}(s,s'') \text{ and } r \not\in T(s).$$

Now $/\,-J/$ gives that $r=r'+r''$ for some $r'$ and $r''$ and furthermore that for some $s'_0$. 

294
From $\mathcal{K}_1(s,s')$ and the expressiveness assumption we get that
\[
\langle c_1, s \rangle \xrightarrow{L_0^1} s_1 \quad \text{for some } r'_1 \text{ and } s_1 \text{ with } s'_1 \epsilon_0 s'.
\]
Lemma 3.1-3 gives that $s'_0 = s'_1$ and $r'=r'_0$. Since $s'_0 \epsilon_0 s'$ and $\text{FV}(c_2) \epsilon_0 V$, we get from Lemma 3.1-2 that for some $s''_0$
\[
\langle c_2, s' \rangle \xrightarrow{L_2^2} s''_0.
\]
The expressiveness assumption gives that $U[c_1](s)=r'$ and $U[c_2](s)=r''$. Since $r=r'+r''$ and $r(T(s))$ we have $r''(U(T-U[c_1]))(s)$ and thereby
\[
P(T-U[c_1])(s,s'). \quad \text{This proves (2).} \]

Case WHILE $b \text{ DO } c$: Assume now that

(1) $\vdash c: T \cup Q / V$

and we shall construct a proof of the formula in $U$. The formulas $P'(z)$, $Q'$ and $Q''$ (and the program $c'$) are defined as in the completeness proof for $\mathcal{I}$, case WHILE $b \text{ DO } c$: $P'(z)$ is the pure formula
\[
\exists z. \exists x'. c : T \cup x', Q' \text{ is } G[c : T \cup x', Q'],
\]
and $Q''$ is $\exists c : T \cup x', Q$. Furthermore, we define the time terms $T'$, $\bar{T}'$ and $T''$ to be $U[c : T \cup x', Q']$, $U[c : T \cup x', Q]$, and $U[c]$, respectively. A straightforward modification of the proof for the similar result in the completeness proof for $\mathcal{I}$, case WHILE $b \text{ DO } c$, shows that the formula
\[
P'(z+1) \odot b \langle c : T'' \rangle P'(z) \odot Q'' / V U[z]
\]
is valid and then, by the induction hypothesis, provable in $U$. Using $/\text{inv-}U / V$ we then get a proof of
\[
P'(z+1) \odot b \langle c : T'' \rangle P'(z) \odot P'(z+1) \odot b \odot Q'' / V U[z].
\]
As in the completeness proof for $\mathcal{I}$, case WHILE $b \text{ DO } c$, we have

295
and from it follows that 

Using Lemma 6.1-1 it is straightforward to prove that \( \mathcal{P}(z+1) \land b \mathcal{Q} \land \mathcal{T}' \). Below we prove that

\[
(2) \quad \mathcal{P}'(z+1) \land b \mathcal{Q} \land \mathcal{T}' \Rightarrow T \land \mathcal{T}'
\]

and since \( U(b) + T^* \Rightarrow T' \) clearly holds we can apply the rule \( \text{WHILE-}U \) and get a proof of

\[
\exists z. \mathcal{P}'(z) \langle \text{WHILE } b \text{ DO } c : T' \rangle \mathcal{Q}' /V.
\]

As in the completeness proof for \( \mathcal{J} \), case \( \text{WHILE } b \text{ DO } c \), we have \( \mathcal{P} \Rightarrow \exists z. \mathcal{P}'(z) \) and \( \mathcal{P} \land \mathcal{Q}' \Rightarrow Q \). Below we prove that

\[
(3) \quad \mathcal{P} \Rightarrow T \land T'
\]

so by applying the rules \( \text{cons-}U \) and \( \text{inv-}U \) we get the required proof of the formula of (1) in \( U \).

To prove (2) assume that \( \mathcal{P}'(z+1) \land b \mathcal{Q} \land \mathcal{T}' \) for some pair \( (s,s') \) of states. Since \( \mathcal{P}'(z+1)(s) \) holds we have for some \( s'' \) and \( r \) that

\[
\langle \text{WHILE } b \text{ DO } c; s \rangle \overset{r}{\rightarrow} s''
\]

and since \( b(s) \) holds \( \text{WHILE-}U \) gives that \( r=b^<(s)+r'+r'' \) and for some \( s_0' \)

\[
\langle c, s \rangle \overset{r_0'}{\rightarrow} s_0' \quad \text{and} \quad \langle \text{WHILE } b \text{ DO } c; s_0' \rangle \overset{r''}{\rightarrow} s''.
\]

From \( \mathcal{Q}'(s,s') \) and the expressiveness assumption we get that

\[
\langle c, s \rangle \overset{s_1'}{\rightarrow} s_1' \quad \text{for some } s_1' \text{ with } s_1' \mathcal{V} s' \text{ and some } r_0'.
\]

Lemma 3.1-3 gives that \( s_0'=s_1' \) and \( r'=r_0' \). Since \( s_0' \mathcal{V} s' \) and \( \text{FV(} \text{WHILE } b \text{ DO } c; s \rangle \mathcal{V} \) we get from Lemma 3.1-2 that

\[
\langle \text{WHILE } b \text{ DO } c; s \rangle \overset{s_0'}{\rightarrow} s_0''
\]

for some \( s_0'' \). The expressiveness assumption gives that

\[
U(\text{WHILE } b \text{ DO } c; s)=b^<(s)+r'+r'' \quad \text{and} \quad U(c; s)=r' \quad \text{and} \quad U(\text{WHILE } b \text{ DO } c; s')=r''.
\]
Lemma 6.1-1 gives that \( U(b)(s) = b^S(s) \) and thereby \( \overrightarrow{T'}(s) = T'(s') \), proving that \( \mathbb{L}(T' \& T')(s, s') \) as required. This proves (2).

To prove (3) assume that \( \mathbb{P}(s) \) holds for some state \( s \). Then (1) gives that for some \( s' \) and \( r \)

\[
\langle \text{WHILE } b \text{ DO } c, s \rangle \rightarrow s', \ \mathbb{Q}(s, s') \text{ and } r \notin T(s).
\]

From the expressiveness assumption we get that \( r = T'(s) \) and thereby \( \mathbb{L}(T' \& T)(s) \). This proves (3).

This completes the proof of The Completeness Theorem for \( \mathbb{U} \).

**FULFILLMENT OF THE STRONG EXPRESSIVENESS CONDITION**

We have now seen that the strong expressiveness condition for \( \mathbb{U} \) is sufficient to ensure that \( \mathbb{U} \) is complete. But unfortunately, it is rather difficult to fulfill this condition for a data type and its computational model.

Let us first consider the data type of Extended Peano Arithmetic and its uniform computational model introduced in Example 3.1-2. By induction on the structure of the terms of the assertion language over this data type it can be proved that any term is bounded by a polynomial in the free variables of the terms, for instance \( 3'x'y+y/x-y/2+3'3'x'y+y+3 \). Clearly, there are while programs that will have for instance exponential time complexity, an example is the program that first computes the exponential function and then in a loop decreases its value by one until it becomes zero. For such a program we cannot have a term in the assertion language for its exact run-time so the strong expressiveness condition for \( \mathbb{U} \) cannot be fulfilled.
The argument above is a special case of a much more general argument showing that it is not sufficient to extend the data type of Extended Peano Arithmetic with function symbols for primitive recursive functions. Then all terms will represent primitive recursive functions but replacing the exponential function in the argument above by Ackerman's function we get a program whose run-time is not primitive recursive.

However, we can get an even stronger result:

**Lemma 6.3-1**: The strong expressiveness condition for \( \mathbb{U} \) is not fulfilled for any extension of the data type of Extended Peano Arithmetic and its uniform computational model with function symbols for (total) recursive functions.

**Proof**: The proof of this lemma rely on the following result /Ro67/:

\[
\begin{aligned}
\text{(§)} & \quad \text{there exists a partial recursive function } f : \mathbb{N} \to \{0, 1\} \text{ that cannot be extended to a (total) recursive function, that is, for every (total) recursive function } g : \mathbb{N} \to \{0, 1\} \text{ there is a natural number } n \text{ such that } f(n) \text{ is defined but } f(n) \neq g(n). \\
\end{aligned}
\]

Any partial recursive function can be computed by a while program so let \( c \) be the program computing the function \( f \) of (§). Without loss of generality we can assume that \( c \) is such that

- \( \langle c, s \rangle \xrightarrow{L} s' \) implies \( z(s') = f(x(s)) \) and \( x(s) = x(s') \),
- if \( \langle c, s \rangle \xrightarrow{r} s' \) and \( x(s) = x(s_0) \) then \( r = r' \),

that is, \( c \) returns the value of \( f \) on \( x \) in \( z \), does not change the value of \( x \) and the run-time of \( c \) only depends on the value of \( x \).
Assuming that the strong expressiveness condition for $\mathcal{U}$ is fulfilled we get that there is a term $\mathcal{U}[c']$ for the exact run-time of the program $c'$ defined by

$$c' \equiv c;c;c;c;\text{IF } z=0 \text{ THEN } z:=0 \text{ ELSE } z:=0+1,$$

that is, if $c'$ terminates then its run-time is given by the term $\mathcal{U}[c']$. In any extension of the computational model the test $z=0$ will require three time units and the assignments $z:=0$ and $z:=0+1$ will require two and four time units, respectively. The program $c$ is such that

$$u_{\mathcal{U}[c']} = \begin{cases} 4'r+5 & \text{for some } r \text{ if } f(x(s))=0 \\ 4'r+7 & \text{for some } r \text{ if } f(x(s))=1. \end{cases}$$

The term $\mathcal{U}[c']$ will represent a (total) recursive function. The function $g: \mathbb{N}\rightarrow\{0,1\}$ defined by

$$g(n) = ((\mathcal{U}[c'](s^n_x) \mod 4)-1)/2 \quad (\text{for any state } s)$$

will then be a (total) recursive function. For any natural number $n$ we will have $g(n)=f(n)$. But this is a contradiction with ($\$) and we have proved the lemma.

6.4 EXAMPLE: DIJKSTRA'S GRAPH ALGORITHM

The results of the previous two sections may seem confusing from a pragmatic point of view. On the one hand, the result of Section 6.2 tells us that we might encounter problems when using the proof system $\mathcal{U}$ that are due to a weakness of the proof system. On the other hand, the result of Section 6.3 seems to suggest that the problems will disappear if we choose a more powerful assertion language. So an interesting question is what these results really mean when we are formalising the traditional informal analyses of
of algorithms.

In this section we shall consider Dijkstra's algorithm for solving the single source shortest path problem /AHU82/. Here is given a directed graph in which each edge has associated a length being a natural number and one of the vertices is specified as the source. The problem is now to determine the length of the shortest path from the source to every other vertex in the graph where the length of a path is the sum of the lengths of the edges on the path.

A directed graph can be represented by an adjacency matrix and an informal analysis of Dijkstra's algorithm using this representation gives an $O(n^2)$ bound on the run-time where $n$ is the number of vertices in the graph. Below we shall see that this bound also can be proved in the proof system $\mathcal{U}$.

Alternatively, we can represent the graph by an adjacency list. An informal analysis of Dijkstra's algorithm now gives an $O(e+n\log(n))$ bound on the run-time. It seems not to be possible to formalise this analysis in the proof system $\mathcal{U}$. The theoretic results suggest that we choose a more powerful assertion language in order to prove the required upper bound but this seems not to give rise to a proper formalisation of the informal analysis. The problem is that the informal analysis uses a book-keeping trick where the run-time requirement is not counted globally but is associated with a number of different accounts reflecting the actual data manipulated by the program. It is not surprising that an analysis using such a trick cannot be formalised properly in a proof system as $\mathcal{U}$ where the proofs are bound to follow the syntax of the actual programs.
Consider now a directed graph $G$ given by a set $V$ of vertices and a set $E (\subseteq V \times V)$ of edges. Let $source(G)$ be a distinguished vertex of $G$ and assume that we have a mapping $l$ that to each edge of $G$ associates a natural number being its length. Dijkstra's algorithm will then find the minimal length of a path from the source vertex to any other vertex in the graph. The idea in the algorithm is to construct a set $S$ of vertices whose minimal distance from the source vertex is already known. A data structure $D$ will contain information about the shortest path from the source vertex to any other vertex passing through vertices of $S$. Using an appropriate data type that will be specified below, the algorithm is as follows:

$$S := \{source(G)\} \subseteq nodes(G);$$
$$D := init(G);$$
$$\text{WHILE } \neg (\text{card}(S) = \text{nodes}(G)) \text{ DO}$$
$$\{$$
$$i := \text{min}(D, S, G);$$
$$S := S \cup \{i\};$$
$$D := \text{upd}(D, i, G)$$
$$\} \quad \text{body}$$

This algorithm is a straightforward reformulation of the pseudo-algorithm that can be found in for instance /AHU82/. The algorithm has been reformulated in order to make the syntactic appearance of the algorithm independent of the choice of the representation of the data structures $G$, $D$ and $S$.

The data type will be an extension of that of Extended Peano Arithmetic (Example 3.1-2). In addition to the sort $\text{nat}$ we have the four sorts $\text{graph}$, $\text{vertex}$, $\text{set-of-vertices}$ and $\text{set-of-distances}$. The new operations are
- source of arity \((\text{graph,vertex})\),
- init of arity \((\text{graph,set-of-distances})\),
- nodes, edges of arity \((\text{graph,nat})\),
- edges of arity \((\text{graph vertex,nat})\),
- \(\{\cdot\}\) of arity \((\text{vertex nat,set-of-vertices})\),
- card of arity \((\text{set-of-vertices,nat})\),
- min of arity \((\text{set-of-distances vertex graph,vertex})\),
- upd of arity \((\text{set-of-distances vertex graph,set-of-distances})\),
- \(\cdot\mathcal{V}\{\cdot\}\) of arity \((\text{set-of-vertices vertex,set-of-vertices})\).

The model \(\mathcal{M}\) we shall consider is an extension of the standard model of the Extended Peano Arithmetic (Example 3.1-2). An element \(g\) of \(\mathcal{M}\) is given by a set \(V_g = \{0, 1, \ldots, n\}\) of vertices, a set \(E_g = \{(V \times V) \subseteq V \times V\}\) of edges and a mapping \(l_g : E_g \rightarrow \mathbb{N}\) associating lengths with the edges. An element \(v\) of \(\mathcal{M}\) is a vertex, that is a natural number and an element \(s\) of \(\mathcal{M}\) will be a finite set of natural numbers. An element \(d\) of \(\mathcal{M}\) is a set of pairs of the form \((v, l)\) where \(v\) and \(l\) both are natural numbers (indeed \(v\) will be thought of as a vertex).

The new symbols of the data type are interpreted as follows

- source\((g)\) = 0,
- init\((g)\) = \(\{(0, 0), (0, v) \mid v \in E_g\}\),
- nodes\((g)\) is the cardinality of \(V_g\),
- edges\((g)\) is the cardinality of \(E_g\),
- edges\((g,v)\) is the cardinality of the set \(\{v' \mid (v,v') \in E_g\}\),
- \(\{v\}_n\) is the singleton set \(\{v\}\),
- card\((s)\) is the cardinality of the set \(s\),
- min\((d,s,g)\) is the minimal \(v\) such that \(v \in V_g\) and for any \(v'\), if \((v,l) \in d\) and \((v',l') \in d\) then \(l \leq l'\),
\[ \text{upd}(d,v,g) = \{(v',1+1_g(v,v'))| (v,v') \in E_g \text{ if } (v',1') \in d \text{ and } 1+1_g(v,v') \in l' \} \]

\[ u\{ (v',1') \} (v',1') \in d \text{ and if } (v,l) \in d \text{ and } (v,v') \in E_g \]

\[ \text{then } 1+1_g(v,v') \in l' \} , \]

- \( \text{su}\{v\} \) is the union of \( s \) and \( \{v\} \).

**A COMPUTATIONAL MODEL**

The run-time requirements of the various operations of the data type may well depend on the implementation we have in mind. We shall here extend the model \( M \) above to a computational model based upon the adjacency matrix representation of a graph.

An element \( g \) of sort **graph** will be represented by an \((n+1) \times (n+1)\) matrix \( M_g \) (where \( V_g = \{0,1,\ldots,n\} \)) such that

\[ M_g[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E_g \\ \infty & \text{otherwise} \end{cases} \]

where \( \infty \) is a new value \((\infty \notin \mathbb{N})\). An element \( v \) of sort **vertex** is represented by a natural number. An element \( s \) of sort **set-of-vertices** is represented by a bit-vector \( B_s \) such that

\[ B_s[i] = \begin{cases} 1 & \text{if } i \in s \\ 0 & \text{otherwise} \end{cases} \]

An element \( d \) of sort **set-of-distances** will be represented by an array \( A_d \) satisfying

\[ A_d[i] = \begin{cases} j & \text{if } (i,j) \in d \\ \infty & \text{otherwise.} \end{cases} \]

Given a data type (and model) with one and two dimensional arrays we can now construct small while programs implementing the
operations of the data type such that the semantic specification of the model $\mathcal{M}$ holds. We shall not go into details here but merely mention that each computational model for the "low level" data type will define a computational model for the "high level" data type.

In order to apply the proof system $\mathcal{U}$ successfully we have to ensure that the ("high level") data type and its computational model satisfy the time term expressiveness condition (confer The Soundness Theorem for $\mathcal{U}$ in Section 6.1). We shall assume that the computational model is such that

$$U^+(x) = 0 \text{ for } x \text{ having sort } \text{nat, graph, vertex, set-of-vertices and set-of-distances}$$

and

- $U_{\text{source}}(G) = 0$,
- $U_{\text{init}}(G) = 2'\text{nodes}(G)$,
- $U_{\text{nodes}}(G) = 0$,
- $U_{\text{edges}}(G) = \text{nodes}(G)'\text{nodes}(G)$,
- $U_{\text{edges}}(G,i) = \text{nodes}(G)$,
- $U_{\text{t1.1}}(i,n) = n$,
- $U_{\text{card}}(S) = 0$,
- $U_{\text{min}}(D,S,G) = 3'\text{nodes}(G)$,
- $U_{\text{upd}}(D,i,G) = 7'\text{nodes}(G)$,
- $U_{\text{ut1.1}}(S,i) = 1$,
- all the operations of the Extended Peano Arithmetic are free.

(In fact this computational model can be obtained from a computational model for the "low level" data type where each reference to an array element requires one unit of time and all other operations are free.)
We can now prove that the run-time of Dijkstra's algorithm is $O(n^2)$ where $n$ is the number of vertices in the graph. More precisely, we shall prove that (for $V=\{S,G,D,i\}$):

$$\text{nodes}(G) \geq 1 \text{Dijkstra:} k_1 \cdot \text{nodes}(G) \cdot \text{nodes}(G) + k_2 \cdot \text{TRUE} / V$$

holds for some constants $k_1$ and $k_2$.

The invariant, $P(z)$, of the while loop is chosen to be

$$(z=\text{nodes}(G) - \text{card}(S)) \land \text{nodes}(G) \geq 1.$$ 

Using the axiom /ass-U/ and the rules /cons-U/ and /inv-U/ we can easily construct proofs of the formulas

$$P(z+1) \land (\text{card}(S) = \text{nodes}(G)) \land i = \min(D,S,G) : 3 \cdot \text{nodes}(G)$$

and

$$P(z) \land \text{nodes}(G) \land S \cup \{i\} : P(z) \land G = \text{GACard}(S) = \text{card}(S) + 1 / V \cup \{z\}$$

So using $\vdash U$ twice and then $\text{cons-U}$ we get a proof of

$$(1) \quad P(z+1) \land (\text{card}(S) = \text{nodes}(G)) \land \text{body:} 10 \cdot \text{nodes}(G) + 1$$

$$P(z) \land G = \text{GACard}(S) = \text{card}(S) + 1 / V \cup \{z\}.$$ 

The next step is to apply the rule $\text{WHILE-U}$. We have

$$\vdash P(0) \rightarrow \text{card}(S) = \text{nodes}(G),$$

$$\vdash P(z) \land \text{card}(S) = \text{nodes}(G) \rightarrow \text{TRUE}$$

and

$$\vdash (G = \text{GACard}(S) = \text{card}(S) + 1) \rightarrow \text{TRUE} \rightarrow \text{TRUE}.$$ 

The term $T$ bounding the run-time of a number of executions of the loop body is chosen to be

305
(10 \cdot \text{nodes}(G) + 1) \cdot (\text{nodes}(G) - \text{card}(S) - 1).

Clearly we have

\[(2) \quad \exists P(z) \land \text{card}(S) = \text{nodes}(G) \rightarrow 0 \cdot T,\]
\[(3) \quad \forall G = \overline{G} \land \text{card}(S) = \text{card}(S) + 1 \rightarrow T \not\in T,\]

and

\[(4) \quad \forall 0 + (10 \cdot \text{nodes}(G) + 1) \lor T \not\in T,\]

so applying /WHILE-\rightarrow/ we get a proof of

\[(5) \quad \exists z. P(z) \land \neg (\text{card}(S) = \text{nodes}(G)) \land \text{body}: T \rightarrow \text{TRUE}/V.\]

It is straightforward to construct a proof of the formula

\[
\exists z. P(z) \land G = \overline{G} \land \text{card}(S) = 1/V,
\]

so using /;\rightarrow/ and /cons-\rightarrow/ we get a proof of

\[
\exists z. P(z) \land \text{Dijkstra: 10 \cdot nodes}(G) \cdot \text{nodes}(G) \rightarrow \text{TRUE}/V,
\]

as required.

Let us briefly compare this analysis with the informal one given in for instance /AHU82/. Here the (informal) analysis starts by observing that the body of the loop is executed $n-1$ times, namely once for each vertex of the graph except the source vertex. The runtime for each execution of the loop is then proved to be $\Theta(n)$ (in essentially the same way as in the formal proof above). So the resulting time requirement of the while loop is $\Theta(n^2)$. The initialising statements of the program take time bounded by $\Theta(n)$ so a resulting upper bound on $\Theta(n^2)$ is obtained.

The number of times the body of the loop is executed does not occur directly in the formal proof above. The formulas (2), (3)
and (4) show that in order to deduce the $\Theta(n^2)$ bound for the while loop in (5) we are calculating a sum of the time requirements. In Section 6.5 we shall see how the formal proof can be modified such that it better formalises the informal analysis.

**Proving the $O(E+N\cdot \log(n))$ Bound in $U$**

In the case where the graph is sparse one will often represent it by an adjacency list rather than an adjacency matrix. A graph with $n$ vertices and $e$ edges will then be represented by two arrays HEAD and EDGE. For each vertex $v$, $\text{HEAD}[v]$ will give an index $i_v$ to the other array EDGE and it will give the number $e_v$ of edges starting in $v$. The entries $\text{EDGE}[i_v], \ldots, \text{EDGE}[i_v+e_v-1]$ will then specify the other endpoint and the length of the edge.

With this representation of the graph the statement $D:=\text{upd}(D,i,G)$ can be performed in time $O(\text{edges}(G,i))$ because we simply have to go through the part of the array EDGE that corresponds to the vertex $i$.

An element of sort set-of-distances can conveniently be represented by a priority queue. The statement $i:=\text{min}(D,S,G)$ can then be performed in time $O(\log(\text{nodes}(G)))$.

An informal analysis of Dijkstra's algorithm goes now as follows. The statements $i:=\text{min}(D,S,G);S:=S\cup\{i\}$ are executed once for each node of the graph and each time it requires a run-time bounded by $O(\log(\text{nodes}(G)))$, giving a total of $O(n\cdot \log(n))$ where $n=\text{nodes}(G)$. The statement $D:=\text{upd}(D,i,G)$ is executed once for each node of $G$ but each time it requires a run-time proportional to the number of edges starting in that node. So the total run-time will be
a vertex \( \text{edges}(G, i) \) which is \( \Theta(e) \) where \( e = \text{edges}(G) \). So the run-time of the while loop is \( \Theta(e + n \cdot \log(n)) \).

The book-keeping trick used here seems very difficult (if not impossible) to formalise in a proof system as \( \mathcal{U} \). It is straightforward to specify a computational model satisfying the time term expressiveness condition of Section 6.1 (however, we shall have to extend the data type with the function symbol log of arity \( \langle \text{nat}, \text{nat} \rangle \) and give it the obvious interpretation). For the body of the loop we can then formally prove the upper bound

\[
k_1 \cdot (\log(\text{nodes}(G)) + \text{edges}(G, \min(D, S, G))) + k_2
\]

for some constants \( k_1 \) and \( k_2 \). In order to apply the while rule we have to find time terms \( T \) and \( T' \) for the loop such that certain conditions are fulfilled. We can choose \( T \) to be

\[
(\text{nodes}(G) - \text{card}(S)) \cdot (k_1 \cdot (\log(\text{nodes}(G)) + \text{edges}(G)) + k_2)
\]

(and \( T' \) accordingly) but this will give a proof of an \( \Theta(n \cdot (e + \log(n)) \) bound on the run-time rather than an \( \Theta(e + n \cdot \log(n)) \) bound.

However, we can extend the data type with a function symbol edge of arity \( \langle \text{graph set-of-vertices}, \text{nat} \rangle \) and define

\[
\text{edge}(g, s) = \sum_{v \in S} \text{edge}(g, v).
\]

In the formal proof we can then let \( T \) be the term

\[
(\text{nodes}(G) - \text{card}(S)) \cdot (k_1 \cdot \log(\text{nodes}(G)) + k_2) + \text{edge}(G, S)
\]

and we will get a proof of the upper bound \( \Theta(e + n \cdot \log(n)) \). But this formal analysis can hardly be said to be a formalisation of the book-keeping arguments of the informal analysis.
6.5 CONCLUDING REMARKS

We have in this chapter specialised the proof system $\mathcal{A}$ of Chapter 3 to prove upper bounds on the run-time of while programs. The resulting proof system $\mathcal{U}$ is not complete in the same sense as the previous proof systems but it can be proved to be complete in some crude sense. On the other hand the proof system $\mathcal{U}$ allows even more natural formalisations of the informal run-time analyses than the previous proof system $\mathcal{A}$.

A number of worked examples have been developed in order to compare the informal analyses of the text books with the formal proofs that can be obtained using a proof system. In Chapter 1 we listed some informal rules for run-time analysis of simple iterative programs suggested by Aho, Hopcroft and Ullman in /AHU82/. So in stead of doing some further examples we shall now discuss how closely these rules have been formalised in the proof systems.

The interesting rule is, of course, that for the while loop. The informal rule of Section 1.2 consists in fact of two rules the first being that

the run-time is the sum, over all times round the loop, of the time to execute the body and the time for evaluating the condition.

This is, indeed, reflected in the rule /WHILE-$\mathcal{U}$/ of the proof system $\mathcal{U}$. The time term $T$ in that rule stands for the time required for executing the body and evaluating the test the remaining number of times. The three hypotheses of the rule

$$P(z) \rightarrow b \rightarrow \mathcal{U}(b) \times T,$$
and
\[ U(b) + T' + \alpha(T) \]

ensure that \( T \) counts the sum of the run-times properly. The informal analysis of the bubble sorting algorithm of Section 3.5 uses the informal rule mentioned above. As the example analysis of Section 6.1 shows we can give a quite satisfactory formalisation of this analysis in the proof system \( \mathcal{U} \).

The second part of the informal rule of Section 1.2 for computing the run-time requirements of the while loop says that

\[ \text{often this time is, neglecting constant factors, the product of the number of times the loop is executed and the maximal run-time for the body.} \]

This is, in fact, the informal rule used in the informal analysis leading to the \( \Theta(n^2) \) bound on the run-time of Dijkstra's algorithm in Section 6.4. As we have noted there the formalisation of the analysis in \( \mathcal{U} \) was not quite satisfactory because of the way the run-time of the while loop was calculated.

However, we can easily construct an alternative while rule that reflects the ideas of the informal rule above more closely:

\[
P(z+1) \vdash b \langle c: T \rangle P(z) \vdash Q'/V \cup \{z\}, \ P(0) \vdash \neg b, \ P(z) \vdash b \vdash I \vdash Q, \ Q' \vdash Q, \\
\text{WHILE}' \vdash \ U(b) = U(b) A \subseteq \mathcal{B} \\
\exists z. P(z) \langle \text{WHILE } b \text{ DO } c: B' (U(b) + T) + U(b) \rangle Q/V
\]

where \( z \) is a variable of sort \( \text{nat} \) satisfying \( z \notin V \).

Here \( B \) is a term (yielding a value of sort \( \text{nat} \)) bounding the number
of times the body of the loop has to be executed. It is straightforward to check that the rule can be derived from the rule /WHILE-\mathcal{U}/ in the proof system \mathcal{U} so the inclusion of /WHILE'-\mathcal{U}/ in \mathcal{U} does not add any additional power to the proof system from a theoretical point of view.

Pragmatically, we do gain something. The informal analysis of Dijkstra's graph algorithm in Section 6.4 leading to the \Theta(n^2) bound can now be formalised quite satisfactorily. As in Section 6.4 we obtain a proof of the formula

\[ P(z+1) \land \neg(\text{card}(S) = \text{nodes}(G)) \langle \text{body}: 10 \cdot \text{nodes}(G)+1 \rangle \]
\[ P(z) \land G = G \land \text{card}(S) = \text{card}(S)+1 / V \forall \{z\}. \]

The term \text{nodes}(G)-\text{card}(S) tells how many times the body of the loop has to be executed yet. Since

\[ G = G \land \text{card}(S) = \text{card}(S)+1 \rightarrow (10 \cdot \text{nodes}(G)+1 = 10 \cdot \text{nodes}(G)+1) \land (0 = 0) \land 
\]
\[ (\text{nodes}(G)-\text{card}(S) < \text{nodes}(G)-\text{card}(S)) \]

we can apply the rule /WHILE'-\mathcal{U}/ and get a proof of

\[ \exists z. P(z) \langle \text{WHILE } \neg(\text{card}(S) = \text{nodes}(G)) \text{ DO body:} \]
\[ (\text{nodes}(G)-\text{card}(S)) \cdot (10 \cdot \text{nodes}(G)+1) \]
\[ \text{TRUE} / V. \]

The rest of the proof is now as in Section 6.4.

The conjecture is therefore that by including the rule /WHILE'-\mathcal{U}/ in the proof system \mathcal{U} we can formalise all the informal analyses of programs that are based on the informal rules of Section 1.2 in a natural way.

An example of an informal analysis that does not follow the informal rules of Section 1.2 is that of Dijkstra's graph algorithm.

311
in Section 6.4 leading to the $O(e^n \log(n))$ upper bound. We have seen that this result can be proved in the proof system $\mathcal{U}$ but as we remarked in Section 6.4, we have not obtained a natural formalisation of the book-keeping argument in the informal analysis. The idea in the book-keeping argument is to associate run-time properties with different accounts representing various pieces of data manipulated by the algorithm. In itself this idea is very estranged from that behind Hoare-like proof systems so it is not surprising that the formalisation is unsuccessfully.

Let us conclude this chapter with a few comments on the completeness results for $\mathcal{U}$. Although The Completeness Theorem for $\mathcal{U}$ proved in Section 6.3 is positive it is also rather weak. However, in practice it seems unlikely that one will encounter problems for that reason (at least if one allows reasonable extensions of the data type with function symbols). An interesting (and open) problem is therefore whether we can obtain a stronger completeness result for $\mathcal{U}$ that, at least, captures most of the cases arising in practice.
The development of the previous chapters shows how Hoare-like proof systems for total correctness can be extended to prove run-time properties of programs. We claim that these proof systems allow quite natural formalisations of existing informal analyses of algorithms and furthermore they have the desired theoretical properties of soundness and completeness (in the sense of Cook).

More specifically, the main achievements of this thesis are

1. We have extended a proof system for total correctness of while programs to prove run-time properties as well. The resulting proof system, said to be in direct style, is such that
   - it is sound and complete (in the sense of Cook),
   - it allows quite natural formalisations of a large class of informal run-time analyses,
   - it does not require the program to be modified by inserting explicit operations upon a clock; furthermore, it compares very well with a method based on such ideas, and
   - a comparison with a couple of other extensions of the total correctness proof system to prove run-time properties shows that the direct style proof system should be preferred.

2. The direct style proof system has been extended to a language containing nested declarations of recursive procedures with
call-by-value parameters. The resulting proof system is such that

- it is sound and complete (in the sense of Cook),

- it allows quite natural formalisations of the traditional informal analyses and, especially, one can obtain recurrence relations in the formal proofs that are rather close to those of the informal analyses, and

- it specifies indirectly a proof system for total correctness of the recursive language; this proof system is sound and complete (in the sense of Cook) and it differs from the existing Hoare-like proof systems for total correctness of recursive procedure programs in that it allows nested declarations of procedures (and this gives rise to the introduction of a new component in the formulas of the proof system).

3. For the while language we have developed a special purpose proof system for proving upper bounds on the run-time. It is such that

- it is sound but not complete in the same sense as the previous proof systems for general run-time properties; however, a weak completeness result can be obtained by imposing a rather crude expressiveness assumption, and

- the proof system allows even more natural formalisations of a large class of informal run-time analyses; however, there are also analyses that seem impossible to formalise in a natural way as proofs in a Hoare-like proof system for run-time analysis (for instance those using a book-keeping trick).
The further work can go in at least three main directions. An obvious possibility will be to continue the work of the chapters 3, 4 and 5 by extending the programming language with new constructs. In textbooks such as /AHU82/ one often analyses pseudo programs rather than programs in some specific language. In order to formalise these analyses in a convenient way it may be helpful to extend the programming language with abstract data types and in this way obtain some of the hierarchical flavour of the informal analyses. Much work has already been done in order to obtain proof systems for partial and total correctness of programs in various languages, for a survey see /Ap81/ and /Ap83/.

Another direction for further work is concerned with the framework for the development of this thesis. The programming language is defined on top of a data type. The meaning of the data type and the specification of its run-time requirements are given by a computational model but it would be interesting to base the development on an axiomatic specification of the data type. Some work has already been done to combine the axiomatic specification of data types with proof systems for partial correctness (see for instance /BeTu81/) and in /AsTu82/ Asveld and Tucker show how an axiomatic specification of a data type can be used to access the complexity of the data and of the operations of the data type.

The third direction for further research is concerned about the development of proof systems for proving orders of magnitude on the worst-case time complexity of programs and is thus a continuation of the work of Chapter 6. We can imagine two steps in this development: first the proof system for upper bounds in Chapter 6 is modified to take the size of input into account and thus to prove upper
bounds on the worst-case time complexity of programs, and secondly, such a proof system is modified to calculate with orders of magnitudes. One of the first problems encountered in such a development is to get a proper notion of the size of input and, unfortunately, there seems not to be any general guidelines neither for how to choose a size measure for a given program nor for what the exact role of this size measure is during an analysis of the worst-case time complexity of the program. When these problems (and others) have been solved and a proof system developed we can return to a discussion of the theoretical properties of soundness and completeness. The results of Chapter 6 already indicates that it may be difficult to obtain a satisfactory completeness result, in fact, we have not obtained a satisfactory result for the upper bound proof system yet.

Finally, we shall mention that even more work is waiting in order to construct proof systems giving natural formalisations of analyses for, for instance, expected time complexity and space complexity of algorithms. Since the flavour of these analyses is rather different from that of worst-case time complexity we shall expect that quite new ideas will be needed in such a development.
APPENDIX A: THE COMPLETENESS PROOF FOR $\mathcal{T}$

The axioms and rules of the proof system $\mathcal{T}$ are given in Section 3.6 and we shall here sketch a proof of the following result:

The Completeness Theorem for $\mathcal{T}$

Given a data type and a numerical computational model for it, if the expressiveness condition for $\mathcal{T}$ is fulfilled then for every well-formed formula $P & R(c > Q & R'/V)$ of $\mathcal{T}$

$T \vdash P & R(c > Q & R'/V)$ implies $T \vdash P & R(c > Q & R'/V)$.  

Furthermore, we shall prove the following result corresponding to Lemma 3.4-1

Lemma: Given a data type containing Peano Arithmetic and given an arithmetical computational model for it, if the exact time term expressiveness condition is fulfilled then so is the expressiveness condition for $\mathcal{T}$.  

Proof of the completeness result for $\mathcal{T}$

The proof of this result is by structural induction on the program $c$. The proof is an extension of that for The Completeness Theorem for $\mathcal{T}$ given in Section 2.4.

Case $x := e$: We assume that

(1) $T \vdash P & R(x := e) Q & R'/V$

holds and we shall construct a proof of the formula in $\mathcal{T}$. From $\text{ass}-\mathcal{T}$ we get a proof of

$P & R(e \text{ time} + \text{U} \{e\}) x := e Q & R'/V$

so using $\text{inv}-\mathcal{T}$ we get a proof of
As in the completeness proof for $\mathcal{F}$, case $x:=e$, we have that

$$
P^\mathcal{F} \rightarrow \forall x \forall y (x := e \rightarrow P^\mathcal{F} = Q)$$

and below we shall prove that

(2) $$P^\mathcal{F} \rightarrow P^\mathcal{F} \rightarrow e \text{time} + U^S(e)$$

so using /cons-$\mathcal{F}$/ we obtain the required proof of the formula of (1) in $\mathcal{F}$.

To prove (2) assume that $P^\mathcal{F}(s,r)$ holds. From (1) we get that for some $s'$ and $r'$

(3) $$\langle x:=e, s, s', r, Q(s,s') \rangle$$

and if $P^\mathcal{F}(s,r''')$ for some $r''$ then $P^\mathcal{F}(s', r'+r'')$.

From /ass-$\mathcal{F}$/, Lemma 3.1-3 and Lemma 3.1-1 we get that $r' = e^S(s) + e(s)^+$ and $s'e^S_x$. Furthermore, Lemma 3.6-1 gives that $r'' = U^S(e)(s)$ so from $P^\mathcal{F}(s,r)$ and (3) we get that $P^\mathcal{F}(s', r + U^S(e)(s))$ (remember that $FV(R') \subseteq \text{V} \cup \text{time}$). But then $P^\mathcal{F}(e \text{time} + U^S(e)(s), r)$ holds and we have proved (2).

Case IF $b$ THEN $c_1$ ELSE $c_2$: Assume that

(1) $$P^\mathcal{F} \land (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) \rightarrow Q^\mathcal{F} \land R' \land /V$$

holds and we shall construct a proof of the formula in $\mathcal{F}$. Define $R''$ to be the formula $\exists \text{time}'. \text{time} = U(b) + \text{time}' \land R' \land \text{time}'$ (that is, $P^\mathcal{F}(s,r)$ holds if and only if $P^\mathcal{F}(s, r + U(b)(s))$ does). Below we shall prove that the formulas

(2) $$P^\mathcal{F}(b \land R'' \land c_1) \rightarrow Q^\mathcal{F} \land R' \land /V$$

and

(3) $$P^\mathcal{F}(b \land R'' \land c_2) \rightarrow Q^\mathcal{F} \land R' \land /V$$

are valid and thereby, using the induction hypothesis, provable in $\mathcal{F}$. From /IF-$\mathcal{F}$/ we then get a proof of the formula.
To prove that the formula (2) is valid assume that $\forall P \forall b(s)$ holds for some $s$. From (1) we then get that for some $s'$ and $r$

$$(4) \quad \langle \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s \text{ s' }, \exists Q(s, s') \rangle \text{ and if } \forall R(s, r) \text{ holds for some } r' \text{ then } \forall R'(s', r'+r) \text{ holds.}$$

Since $\forall b(s)$ holds we get from $\langle \text{IF-} Q' \rangle$ that $r=b^S(s)+r'$ for some $r'$, and that

$$\langle c_1, s \rangle e s'. $$

Assume now that $\forall R''(s, r'')$ holds for some $r''$. This means that

$\forall R(s, r'') - U(b)(s)$ holds and thereby that $\forall R'(s', r'' - U(b)(s)+r)$ holds (according to (4)). Since $U(b)(s)=b^S(s)$ (Lemma 3.6-1) and $r=b^S(s)+r'$ we get $\forall R'(s', r''+r)$ as required. This proves the validity of (2).

The proof showing that (3) is valid is similar and therefore omitted.

Case $c_1;c_2$ : Assume that

$$ (1) \quad \forall P \& \forall c_1; c_2 \rightarrow \exists Q \& \exists R'/V$$

holds and we shall construct a proof of the formula in $T$. As in the completeness proof for $T$, case $c_1;c_2$, we define the formulas $Q_1$, $Q_2$ and $P'$ to be $G[c_1] \land \forall_{V,FV}(c_1)$, $G[c_2] \land \forall_{V,FV}(c_2)$, and $\exists X', G[c_2]^X_{X'}$ (where $X'$, $X$ and $X'$ are "as usual"), respectively. Furthermore, we define the time formulas $R_1$ and $R_2$ to be $WP[c_1,R_2]$ and $WP[c_2,R_1]$, respectively. Using the expressiveness assumption for $T$ (and the lemmas 3.1-3 and 3.1-1) it is straightforward to prove that the...
two formulas

\[ \text{P&R}_1(c_1, P'Q_1 & R'_2) / V \]

and

\[ \text{P'}&R_2(c_2, Q_2 & R'/V) \]

are valid. So the induction hypothesis gives that they are provable in \( \mathcal{T} \) and using \( /; \mathcal{T}/ \) we get a proof of

\[ \text{P&R}_1(c_1, c_2, Q_1, Q_2 & R'/V). \]

As in the completeness proof for \( \mathcal{T} \), case \( c_1, c_2 \), we have \( Q_1, Q_2 \rightarrow Q \).

Below we prove that

(2) \( \text{P&R} \rightarrow \text{P&R}_1 \).

Using first \( /\text{inv-}\mathcal{T}/ \) and then \( /\text{cons-}\mathcal{T}/ \) we now get a proof of the formula of (1).

To prove (2) assume that \( \text{P&R}(s, r) \) holds for some pair \((s, r)\).

Since \( \text{P}(s) \) holds we get from (1) that for some \( s' \) and \( r' \)

(3) \( \langle c_1, c_2, s \rangle \Rightarrow s', Q(s, s') \) and if \( R(s, r') \) then \( R'(s', r + r') \).

The semantic rule \( /; -\mathcal{S}/ \) gives that \( r' = r'_1 + r'_2 \) and that for some \( s'' \)

\( \langle c_1, s \rangle \Rightarrow_1 s'' \) and \( \langle c_2, s'' \rangle \Rightarrow_2 s' \).

Since \( R(s, r) \) holds we get from (3) that \( R'(s', r + r'_1 + r'_2) \) holds. But then the expressiveness condition for \( \mathcal{T} \) gives that \( R_2(s'', r + r'_1) \) and then \( R_1(s, r) \) holds. This proves (2).

Case WHILE \( b \) DO \( c \): Assume that

(1) \( \text{P&R}(\text{WHILE } b \ \text{DO } c) \rightarrow \text{Q&R}'/V \)

holds and we shall construct a proof of the formula in \( \mathcal{T} \). As in the completeness proof for \( \mathcal{T} \), case WHILE \( b \ \text{DO } c \), we define \( P'(z) \) to be
the formula $\exists z'. \exists x'. G[\text{WHILE } b \text{ DO } c'][x']$ (where $z'$, $x'$, $\overline{x}$, $x'$ and $c'$ are as before). The relational formulas $Q'$ and $Q''$ are defined to be $G[\text{WHILE } b \text{ DO } c'[\text{V-FV(WHILE } b \text{ DO } c')]$ and $\exists \text{AG}[c'[\text{V-FV(c')}], \text{respectively. Finally, the formula } R'' \text{ is defined to be }$

$$\exists \text{time}'. \text{time}=\text{U}(b)+\text{time}' \text{WP}[\text{WHILE } b \text{ DO } c', R'] \text{time}' .$$

Below we shall prove that the formula

$$(2) \quad P'(z+1) & R'' \text{c}; P'(z) Q'' & R'' \text{time}',$$

is valid, and thereby, using the induction hypothesis, provable in $\mathcal{T}$. As in the completeness proof for $\mathcal{T}$, case WHILE $b$ DO $c$, we have $FP'(0) \rightarrow b$, $FP'(z) \text{bAR} \rightarrow Q' $ and $FP'' Q' \rightarrow Q' $. Below we shall prove that

$$(3) \quad FP'(z) \text{bAR} \rightarrow R'$$

so we can apply the rule $/\text{WHILE-}\mathcal{T}/$ and get a proof of

$$\exists z. P'(z) \text{AR} \text{time}+\text{U}(b) \text{WHILE } p \text{DO c'} \text{AR}', \text{time}$$

As in the completeness proof for $\mathcal{T}$, case WHILE $b$ DO $c$, we have that $FP \rightarrow \exists z. P'(z) $ and $FAQ' \rightarrow Q$. We shall prove that

$$(4) \quad FPAR \rightarrow PAR \text{time}+\text{U}(b) \text{time}$$

so applying first $/\text{cons}-\mathcal{T}/$, then $/\text{inv}-\mathcal{T}/$ and finally $/\text{cons}-\mathcal{T}/$ we get the required proof of the formula of (1).

To prove (2) assume that $FP'(z+1) \text{b}(s)$ holds for some $s$. As in the completeness proof for $\mathcal{T}$, case WHILE $b$ DO $c$, we can prove that there is a state $s'$ and a natural number $r$ such that

$$\langle c, s \rangle \text{AR} \text{b}s' \text{ and } FP'(z) Q'' (s, s').$$

So assume now that $FR''(s, r')$ holds for some $r'$, that is

$$\text{WPW}[\text{WHILE } b \text{ DO } c \text{U}(s, r' - U(b)(s))].$$

According to the expressiveness
condition for $\bar{f}$ this means that for some $s''$ and $r''$

$$\langle \text{WHILE } b \text{ DO } c, s' \rangle \rightarrow s'' \text{ and } \forall R'(s'', r' - U(b)(s) + r'').$$

Since $b(s)$ holds we get from $\forall/b/ \forall/WHILE$ that $r'' = b^S(s) + r''_1 + r''_2$ and for some $s'_0$

$$\langle c, s' \rangle \rightarrow s' \text{ and } \langle \text{WHILE } b \text{ DO } c, s'_0 \rangle \rightarrow s''.$$

From the lemmas 3.1-3 and 3.1-1 we get that $s'_0 \forall/s' \forall/s''$ and $r''_1 = r$. The expressiveness assumption gives that

$$\forall WP[\text{WHILE } b \text{ DO } c, R'(s_0', r' - U(b)(s) + r' - r'')].$$

holds. We have $U(b)(s) = b^S(s)$ (Lemma 3.6-1) and $FV(WP[\text{WHILE } b \text{ DO } c, R']) \subseteq \forall/\forall/time$ so we get $WP[\text{WHILE } b \text{ DO } c, R'(s', r' + r')$ and thereby

$$\forall R'^{time+U(b)(s', r' + r')} \text{ as required. This proves the validity of (2).}$$

To prove (3) assume that $P'(z) \forall/b/ \forall R'(s, r)$ for some pair $(s, r)$. The expressiveness assumption gives that for some $s'$ and $r'$

$$\langle \text{WHILE } b \text{ DO } c, s' \rangle \rightarrow s' \text{ and } \forall R'(s', r' - U(b)(s) + r').$$

Since $b(s)$ holds we get from $\forall/b/ \forall/WHILE$ that $s = s'$ and $r' = b^S(s)$ (using Lemma 3.1-3). Lemma 3.6-1 gives that $U(b)(s) = b^S(s)$ so we get $R'(s, r)$ as required. This proves (3).

To prove (4) assume that $P \forall R(s, r)$ holds for some pair $(s, r)$. From the assumption (1) we then get that for some $s'$ and $r'$

$$\langle \text{WHILE } b \text{ DO } c, s' \rangle \rightarrow s', \forall Q(s, s') \text{ and if } R(s, r'') \text{ holds for some } r'' \text{ then } R'(s', r' + r') \text{ holds.}$$

Since $R(s, r)$ holds we thus have $R'(s', r + r')$ and the expressiveness assumption gives that $R''(s', r)$ as required. This proves (4). ///

This completes the proof of The Completeness Theorem for $\bar{f}$. 322
Proof of the lemma

We shall prove that the formulas $G\langle c \rangle$ and $WP\langle c, R \rangle$ exist for every while program $c$ and time formula $R$. The formulas $G\langle c \rangle$ are constructed as in the proof for Lemma 2.4-1 so we omit the details here. In order to construct the formulas $WP\langle c, R \rangle$ we shall construct another formula $E^*[c]$ with $FV(E^*[c]) \subseteq FV(c) \cup \{time, time\}$ where $time$ is a special variable of sort $nat$. We shall then define $WP[c, R]$ to be

$$\exists \bar{x}.3\text{time}' . E^*[c]\bar{x} . time' time \wedge \bar{x} . time'$$

where $x, \bar{x}$ and $x'$ are "as usual" and $time'$ is a new variable of sort $nat$. The formulas $E^*[c]$ are defined structurally as follows:

$$E^*[x:=e] \equiv I_{FV(e)} - \{x\} \wedge x = e \wedge time = time' \wedge \bar{S}(e)$$

$$E^*[\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2] \equiv$$

$$((b \in E^*[c_1] \wedge V_{FV(c_1)}) \wedge (\neg b \in E^*[c_2] \wedge V_{FV(c_2)})) \wedge \text{time} + U(b)$$

where $V = FV(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2)$

$$E^*[c_1;c_2] \equiv \exists \bar{x}'.3\text{time}'. E^*[c_1]\bar{x}' . time' time \wedge \bar{x}'$$

$$E^*[c_2]\bar{x} . time' time \wedge \bar{x}$$

where $V = FV(c_1;c_2)$

$$E^*[\text{WHILE } b \text{ DO } c] \equiv$$

$$\exists n.3x.3y.(\theta(X, time, n, Y, y) \wedge \overline{\theta(X, time, 0, Y, y)} \wedge \bar{b} \wedge$$

$$\forall i.3x'.3y'.(i < n \wedge \theta(X', time', i, Y, y) \rightarrow$$

$$\exists x''.3\text{time}''.E^*[c]\bar{x} . x'' \wedge time''. time' \wedge$$

$$\forall x'.3\text{time}'.(i < n \wedge \theta(X', time', i, Y, y) \rightarrow$$

$$\exists x''.3\text{time}''.3\text{time}'.(i < n \wedge \theta(X', time', i, Y, y) \rightarrow$$

The formula $\theta(X, time, i, Y, y)$ used above is defined as in the proof
of Lemma 3.4-1. We omit the proof showing that the required properties hold for the formulas $WP_{c,R}$ defined in this way.
APPENDIX B: PROOFS OF PROPERTIES OF THE NON-RECURSIVE
PROCEDURE LANGUAGE

The semantics and the run-time requirements of the non-recursive procedure language is given by the formal system $S^N$ in Section 4.1. In that section we also list some properties holding for the semantics and run-time requirements of the programs and in this appendix we shall sketch the proofs of four of these results.

The first of the results, Lemma 4.1-3, expresses that we can extend the V-part of a formula $(V,env)\vdash <c,s>_v \rightarrow s'$ to specify some further program variables:

**Lemma 4.1-3:** If $(V,env)\vdash <c,s>_v \rightarrow s'$ and $V' \cap V = \emptyset$ then for some $s''$

$$(V\cup V',env)\vdash <c,s>_v \rightarrow s''$$

where $s'' \in V'$ and $s'' \notin V',s'$.  

In the proof of this result we shall use the following lemma for renaming of variables:

**Lemma 4.1-4:** Assume that $(V,env)\vdash <c,s>_x^{y(s)} \rightarrow s'$ and $x$ and $y$ have the same sorts and satisfy $x \in V$ but $y \notin V$. Then for some $s''$

$$(V\cup \{y\},env)\vdash <c_x^y,s>_x \rightarrow s''$$

where $s'' \in V\cup \{y\}$, $x(y)$ and $y(x')$ and the two proofs in $S^N$ have the same lengths.

The third lemma from Section 4.1 that we shall prove here expresses that the procedure language in deterministic:

**Lemma 4.1-5:** If $(V,env)\vdash <c,s>_v \rightarrow s'$ and $(V,env)\vdash <c,s>_v \rightarrow s''$ then $s' \in V\cup V'$ and $s'' \notin V\cup V'$.  

In order to prove this result we shall use a lemma that allows us to rename procedures:


325
Lemma 4.1-6: Assume that $(V, env) \vdash <c, s> \xrightarrow{\ell} s'$, $p \in \text{DOM}(env)$ and $q \notin \text{DOM}(env)$. Then $(V, env^q_p) \vdash <c^q_p, s> \xrightarrow{\ell} s'$ and the two proofs in $\mathcal{G}^N$ have the same lengths.

Below we sketch the proofs of these four lemmas.

Proof of Lemma 4.1-3

The proof is by induction on the length of the proof of $(V, env) \vdash <c, s> \xrightarrow{\ell} s'$ in $\mathcal{G}^N$. Only two cases are interesting, namely those where either /LET-$\mathcal{G}^N$/ or /CALL-$\mathcal{G}^N$/ is the last rule applied in the proof.

Case /LET-$\mathcal{G}^N$/: Assume now that we have got a proof of

$$(V, env) \vdash \langle \text{LET } x = e \text{ IN } c, s \rangle \xrightarrow{e(s)+e(s)+r} s'$$

from

$$(1) \quad (V\cup\{y\}, env) \vdash \langle c_x, s_y \rangle \xrightarrow{\varepsilon_\gamma} s'$$

where $y$ has the same sort as $x$ and satisfies $y \notin V$. We have two cases.

If $y \notin V'$ then $V'\cap (V\cup\{y\}) = \emptyset$ and the induction hypothesis can be applied to (1) and gives

$$(V\cup V', env) \vdash \langle c_x, s_y \rangle \xrightarrow{e(s)+e(s)+r} s''$$

for some $s''$ where $s'' \notin V\cup\{y\}$ and $s'' \notin V\cup\{y\}$. Then /LET-$\mathcal{G}^N$/ gives

$$(V\cup V', env) \vdash \langle \text{LET } x = e \text{ IN } c, s \rangle \xrightarrow{e(s)+e(s)+r} s''$$

and clearly we have $s'' = s'$ and $s'' = s$.

In the other case we have $y \in V'$. Let now $y'$ be a variable of the same sort as $y$ (and $x$) satisfying $y' \notin V\cup V'$. Let $s_0$ be the state $s_{y'}^{e(s)}$.

Then $s_0y^{\varepsilon_\gamma} \in V\cup\{y\}$ so using Lemma 4.1-2 we get from (1)

$$(V\cup\{y\}, env) \vdash \langle c_x, s_{0y}^{y'(s_0)} \rangle \xrightarrow{\varepsilon_\gamma} s_0'$$
where \( s''_0 = \nu y y' \} s' \). Lemma 4.1-4 then gives

\[
(\forall y' y', \text{env}, y') \vdash c^y y' y, s_0 \to s''_0
\]

where \( s''_0 = \nu y y' y' s' \). Now \( y \notin \text{FV}(c^y y') \) and \( y \notin \text{FV} \) so from Lemma 4.1-2 we get

(2) \( (\forall y' y', \text{env}, y') \vdash c^y y' y, s_0 \to s''_0 \)

where \( s''_0 = \nu y y' y' s''_0 \). We have \( c^y y' \equiv c^y x \) and since \( \text{FV} \) we also have \( \text{env} \) and \( \text{env} \). So (2) really means

(3) \( (\forall y' y', \text{env}, y') \vdash c^y y' e(s) \to s''_0 \).

We have the following calculations \( s''_1 = s', s''_2 = s''_0 \). We shall now apply the induction hypothesis to (3) (note the proofs of (1) and (3) have the same lengths) and get

\[
(\forall y' y', \text{env}, y') \vdash c^y y' e(s) \to s''_2
\]

where \( s''_2 = s''_1 = s''_0 \). From \( \text{LET}^N \) we get

\[
(\forall y' y', \text{env}, y') \vdash \text{LET}_{x=e} \text{IN} c, e \to s''_2.
\]

We have \( s''_2 = s''_1 = s''_0 \) so the result follows.

**Case /CALL-ON/:** Assume now that we have got a proof of

\[
(\forall \text{env}, y, p(e, y), s) \vdash s''_2
\]

from

(1) \( (\forall \text{env}, y, p(e, y), s) \vdash s''_2 \)

where \( \text{env}(p) = (x', y', c) \) and \( x \) is a variable of the same sort as \( x' \) satisfying \( x \notin \text{V} \). The proof is very similar to that above. If \( x \notin \text{V} \) we can apply the induction hypothesis to (1) directly and the required result follows by applying the rule /CALL-ON/. If \( x \in \text{V} \) we shall rename \( x \) to \( x'' \) where \( x'' \) is a variable of the same sort as \( x \) (and \( x' \)) satisfying \( x'' \notin \text{V} \) and then apply the induction.
hypothesis. The proof is very similar to that in the case /LET-\textit{f}^N/ above so we shall omit the details. 

Proof of Lemma 4.1-4

The proof of this result is by structural induction on the length of the proof of \((V, \text{env}) \vdash \langle c, s_x^y(s) \rangle \rightarrow_s^t s\) in \(\gamma^N\). Depending on the last axiom or rule applied we have the following cases:

Case /ass-\textit{f}^N/: We then have

\[
(V, \text{env}) \vdash \langle x' := e, s_x^y(s) \rangle \rightarrow_{x'} s_x^y(s) v
\]

where \(r = e_x^y(s) + e_x^y(s)^+\) and \(v = e_x^y(s)\). We have \(e_x^y(s) = v\) and \((e_x^y(s) + e_x^y(s)^+) = r\). If \(x' \nmid x\) then \((x' := e)_x^y e_x^y = e_x^y\) and using /ass-\textit{f}^N/ we get

\[
(Vuvy_1, \text{env}_x^y) \vdash \langle y := e_x^v, s_x^v \rangle \rightarrow_{x} s_x^v_y.
\]

The required result follows since \(s_x^y \equiv Vuvy_1 (s_x^y v, x(s) v, y(s) v, y)\).

In the case where \(x \mid x'\) we have \((x' := e)_x^y e_x^y = e_x^y\) and /ass-\textit{f}^N/ gives

\[
(Vuvy_2, \text{env}_x^y) \vdash \langle x' := e_x^v, s_x^v \rangle \rightarrow_{x'} s_x^v_x,
\]

The result follows since \(s_x^y \equiv Vuvy_2 (s_x^y v, x(s) v, x(s) v, y)\).

Case /IF-\textit{f}^N/: The proof is straightforward using the induction hypothesis so we omit the details.

Case /i-\textit{f}^N/: Assume that we have a proof of

\[
(V \text{ env}) \vdash \langle c_1; c_2, s_x^y(s) \rangle \rightarrow_{r_1 r_2} s''
\]

from

\[
(V, \text{env}) \vdash \langle c_1, s_x^y(s) \rangle \rightarrow_{r_1} s'.
\]

and
(2) \((V, \text{env}) \vdash \langle c_2, s' \rangle \Rightarrow s''\).

The induction hypothesis applied to (1) gives that

(3) \((V \cup \{y\}, \text{env}^y_x) \vdash \langle c_y, s_0 \rangle \Rightarrow^y s'_1\)

where \(s'' \equiv \{V \cup \{y\}\} x(s) x(s')\). The next step is now to apply Lemma 4.1-2 to (2). We have

\[
\begin{align*}
\forall \nu \delta \gamma & \quad \forall \nu \delta \gamma \quad \forall \nu \delta \gamma \\
\forall \nu \delta \gamma & \quad \forall \nu \delta \gamma \\
\forall \nu \delta \gamma & \quad \forall \nu \delta \gamma
\end{align*}
\]

where \(s'' \equiv \{V \cup \{y\}\} x(s) x(s')\). Since the two proofs have the same length we can apply the induction hypothesis and get

\[
\begin{align*}
(V \cup \{y\}, \text{env}^y_x) & \vdash \langle c_y, s_0 \rangle \Rightarrow^y s'_1 \\
& \quad \forall \nu \delta \gamma \quad \forall \nu \delta \gamma \\
& \quad \forall \nu \delta \gamma \\
& \quad \forall \nu \delta \gamma \\
& \quad \forall \nu \delta \gamma
\end{align*}
\]

The following calculations show that \(s'' \equiv \{V \cup \{y\}\} x(s) x(s')\):

\[
\begin{align*}
\forall \nu \delta \gamma & \quad \forall \nu \delta \gamma \\
\forall \nu \delta \gamma & \quad \forall \nu \delta \gamma \\
\forall \nu \delta \gamma & \quad \forall \nu \delta \gamma \\
\forall \nu \delta \gamma & \quad \forall \nu \delta \gamma
\end{align*}
\]

Case \(\text{WHILE-}^y\): In the case where the conditional is true we proceed essentially as above in the case \(\vdash^y\). The case where the conditional is false is straightforward. We omit the details.

Case \(\text{LET-}^y\): Assume now that we have obtained a proof of

\[
(V, \text{env}) \vdash \langle \text{LET } x' = e \text{ IN } c \rangle \Rightarrow s''
\]

(where \(x' = s(y(s)) + e(s(y(s))\)) from

\[
(V \cup \{x''\}, \text{env}) \vdash \langle c', s_y(v(s), v, y(s')) \Rightarrow s'\)
\]

where \(v = e(s_y(s))\) and \(x''\) is a variable of the same sort as \(x'\), satisfying that \(x'' \notin V\). Assume now that \(x'' \notin V\). Then \(s_y(v(s), v, y(s'))\) and since \(y \notin V\) we get from the induction hypothesis that

(2) \((V \cup \{x''\}, \text{env}^y) \vdash \langle c_2, s_0 \rangle \Rightarrow s''\).
where (using that \( x(s_{x}) = x(s) \)) \( s \models \psi \cup \{ y \} \mid x(s) x(s') \). We have

\[ e^y_{x}(s) = v \text{ and } (e^y_{x})_{s+x}(s) + e^y_{x}(s) = r' \]  

We now have three cases:

**Case 1:** \( x \not\in x' \) and thereby \((\text{LET } x' = e \text{ IN } c)^y_{x} \not\in \text{LET } x = e^y_{x} \text{ IN } c \). We have \( c^x_{x}, y \models c^x_{x} \) and from (2) and the semantic rule /LET-\( ^{y_{x}} \)/ we get

\[ (\psi \cup \{ y \}, x_{x}^y \vdash \text{LET } x = e^y_{x} \text{ IN } c, s_{x}, r_{x} \mapsto \text{LET } x = e^y_{x} \text{ IN } c, s_{x} \)  

We clearly have \( s \models \psi \cup \{ y \} \mid x(s) x(s') \).

**Case 2:** \( x \not\in x' \) and \( y \not\in x' \). Then \((\text{LET } x' = e \text{ IN } c)^y_{x} \not\in \text{LET } x = e^y_{x} \text{ IN } c^y_{x} \) and we have \( c^x_{x}, y \models c^x_{x} \). So from (2) we get, using /LET-\( ^{y_{x}} \)/

\[ (\psi \cup \{ y \}, x_{x}^y \vdash \text{LET } x' = e^y_{x} \text{ IN } c^y_{x}, s_{x}, r_{x} \mapsto \text{LET } x' = e^y_{x} \text{ IN } c^y_{x}, s_{x} \]  

and we also have \( s \models \psi \cup \{ y \} \mid x(s) x(s') \).

**Case 3:** \( x \not\in x' \) and \( y \not\in x' \). Then \((\text{LET } x' = e \text{ IN } c)^y_{x} \not\in \text{LET } x' = e^y_{x} \text{ IN } c^y_{x} \) where \( y' \) is a variable of the same sort as \( y \) satisfying \( y' \notin \text{FV}(c) \cup \text{FV}(e) \).

We have \( c^x_{x}, y \models y' x '' \) and from (2) we get

\[ (\psi \cup \{ y \}, x_{x}^y \vdash \text{LET } y' = e^y_{x} \text{ IN } c^y_{y}, s_{x}, r_{x} \mapsto \text{LET } y' = e^y_{x} \text{ IN } c^y_{y}, s_{x} \]  

and we also have \( s \models \psi \cup \{ y \} \mid x(s) x(s') \).

Assume now that \( x \not\in y \). Then \( s \not\models \\psi \cup \{ y \} \mid x(s) x(s') \) \( v \) \( v \) for \( y' \not\in \psi \cup \{ y \} \)

so from (1) and Lemma 4.1-2 we get

\[ (3) (\psi \cup \{ y \}, y', \psi, y, \psi, y, v, v) \mapsto s_{0} \]  

where \( s_{0} \models \psi \cup \{ y \} \). Since \( y' \mid y \models v \) and the proofs of (1) and (3) have the same length we get from the induction hypothesis

\[ (4) (\psi \cup \{ y, y \}, y', y, \psi, y, \psi, y, v) \mapsto s_{0} \]  

where (using that \( y \mid y, y' \models v \) and \( y \mid y \models y(s) \)) \( s_{0} \models \psi \cup \{ y, y \} \) \( y \models y' \). Now \( c_{x}^y y' \models c_{x}^y y' \) and \( y' \not\in \text{FV}(c_{x}^y) \cup \text{FV}(\psi) \) so we can apply Lemma 4.1-2 to (4) and get

330
(5) \( (V\cup y')^x, env) \vdash \langle c^x, y(s), x^y, s_x^y \rangle \Rightarrow_{y^y} s'

(\text{using } y^y_{FV(env)\cup V}) \text{ where } s''_{\cup y} y'_{x} s''_{0}(\cup y'_{x} s''). \text{ We now proceed as above in the case where } x'' \neq y. \text{ Note that the proofs of (1) and (5) have the same lengths so the induction hypothesis can be applied.}

**Case \(/\text{PROC}^{-}\)**: Assume now that we have a proof of

\[ (V, env) \vdash \langle \text{PROC } p(\text{VAL } x', \text{VAR } y') \text{ IS } c_1 \text{ IN } c_2, s_x^y(s) \rangle \Rightarrow_{y^y} s', \]

from

\[ (V, env(q=(x', y', c_1))) \vdash \langle c^q, s_x^y(s) \rangle \Rightarrow_{y^y} s', \]

where \( q \) is a procedure name satisfying \( q \in \text{DOM(env)} \). The induction hypothesis gives

\[ (V\cup y', env(q=(x', y', c_1))) \vdash \langle c^q, y(s), x(s), x(s') \rangle \]

where \( s''_{\cup y} y'_{x} s''_{x} x' x'' \). The definition of \( env(q=(x', y', c_1)) \) is such that we from \(/\text{PROC}^{-}\) get

\[ (V\cup y'), env_{x} \vdash \langle \text{PROC } p(\text{VAL } x', \text{VAR } y') \text{ IS } c_1 \text{ IN } c_2, s_x^y(s) \rangle \Rightarrow_{y^y} s', \]

and the result follows.

**Case \(/\text{CALL}^{-}\)**: Assume now that we have obtained a proof of

\[ (V, env) \vdash \langle \text{CALL } p(e, y''), s_x^y(s) \rangle \Rightarrow_{y''} s', \]

from

(1) \( (V\cup x'', y') \vdash \langle c^x, y''(s), x^y, s_x^y \rangle \Rightarrow_{y^y} s', \)

where \( env(p)=(x', y', c) \), \( x'' \) is a variable of the same sort as \( x' \) satisfying \( x'' \neq y'. \text{ First assume that } x'' \neq y. \text{ Since } s_x^y(s) \Rightarrow_{y^y} y(s) \text{ we get from the induction hypothesis that}

(2) \( (V\cup x'', y') \vdash \langle c^x, y''y, x^y, s_x^y \rangle \Rightarrow_{y^y} s'' \)

where (using that \( x(s_x^y)=x(s) \)) \( s''_{\cup y} y'_{x} s''_{x} x' x'' \). We have
\[ e_x(s) = v \quad \text{and} \quad (e_x)'(s) + e_x(s) = r' \]. We have three cases:

**Case 1:** \( x \notin \{x', y'\} \). Then env \( (p) = (x', y', c) \) and either \( c^{x''y''} \in c^{x'y'} \) (if \( x \equiv y' \)) or \( c^{x''y''} \in c^{x'y'} \) (if \( x \not\equiv y' \)). Using \( \text{CALL-}\mathcal{S}_N \) we then get

\[ \forall x, y \in \mathcal{S}_N \langle \text{CALL} p(e_x, y, s'), s' \rangle = r' + r \to s' \]

**Case 2:** \( x, y \in \{x', y'\} \). Then env \( (p) = (x', y', c_y) \) and we have either \( c^{x''y''} \in c^{x'y'} \) (if \( x \equiv y' \)) or \( c^{x''y''} \in c^{x'y'} \) (if \( x \not\equiv y' \)) and the result follows using \( \text{CALL-}\mathcal{S}_N \).

**Case 3:** \( x \in \{x', y'\} \) but \( y \notin \{x', y'\} \). Then env \( (p) = (x_0', y_0', c_{x'y'y'}) \)

where \( x_0' \) and \( y_0' \) are distinct variables of the same sort as \( x' \) and \( y' \), respectively, and \( x', y' \in \text{FV}(\text{env}) \cdot \text{FV}(c) \). We now have that either \( c^{x''y''} \in c_{x'y'y'} \) (if \( x \equiv y' \)) or \( c^{x''y''} \in c_{x'y'y'} \) (if \( x \not\equiv y' \)) and in both cases the result follows using \( \text{CALL-}\mathcal{S}_N \).

In the case where \( x'' \equiv y \) we first replace \( x'' \) by \( y' \) in (1) very much as in the case \( \text{LET-}\mathcal{S}_N \) and then we proceed as above. We omit the details.

//

**Proof of Lemma 4.1-5**

We proceed by induction on the length of the proof of \( (V, \text{env}) \langle \langle c, s \rangle \to s' \rangle \). The cases where the last axiom or rule of \( \mathcal{S}_N \) that has been applied is one of \( \text{ass-}\mathcal{S}_N \), \( \text{IF-}\mathcal{S}_N \), \( \text{WHILE-}\mathcal{S}_N \) and \( \text{SWITCH-}\mathcal{S}_N \) are straightforward and will therefore be omitted. The remaining three cases are as follows:

**Case \( \text{LET-}\mathcal{S}_N \):** We have got a proof of

\[ (V, \text{env}) \langle \text{LET} x = e \quad \text{IN} \quad c, s \rangle \]

\[ \text{to get} \quad s' \]
from

(1) \((V \cup \{x\}, \text{env}) \vdash \langle c^{x'}_x, s^{e(s)}_{x'} \rangle \rightarrow s'\)

where \(x' \notin V\). We have a proof of

\(\langle V, \text{env} \rangle \vdash \langle \text{LET } x = e \text{ IN } c, s \rangle \rightarrow s''\)

in \(\mathbb{S}^N\) and it can only be obtained by using \(\text{LET-}\mathbb{S}^N\). So we will have a proof of a formulas of the form

(2) \((V \cup \{x''\}, \text{env}) \vdash \langle c^{x''}_x, s^{e(s)}_{x''} \rangle \rightarrow s''\)

where \(r = e^\mathbb{S}(s) + e(s)^+ + r''\). If \(x'' \in x''\) then the induction hypothesis can be applied to (1) and (2) and we get \(e = r''\) and \(s'' \in \mathbb{S}^V\) and thereby \(e^\mathbb{S}(s) + e(s)^+ + r'' = e''\) and \(s'' \in \mathbb{S}^V\).

So assume now that \(x'' \notin x''\). We shall then use Lemma 4.1-4 and rename \(x''\) to \(x'\) in (2). Let \(s_0 = s^{e(s)}_{x'}\). Then \(s''_{x'} \in \mathbb{S}^V\) and Lemma 4.1-2 applied to (2) gives

\((V \cup \{x''\}, \text{env}) \vdash \langle c^{x''}_x, s^{e(s)}_{x''} \rangle \rightarrow s''\)

where \(s'' \in \mathbb{S}^V\). Since \(x'' \not\in V \cup \{x''\}\) we can apply Lemma 4.1-4 and get

\((V \cup \{x''\}, \text{env}) \vdash \langle c^{x''}_x, s^{e(s)}_{x''} \rangle \rightarrow s''\)

where \(s''_{x'} \in \mathbb{S}^V\) and Lemma 4.1-2 can be applied again and gives

\((V \cup \{x''\}, \text{env}) \vdash \langle c^{x''}_x, s^{e(s)}_{x''} \rangle \rightarrow s''\)

where \(s''_{x'} \in \mathbb{S}^V\). We can now proceed as above in the case where \(x'' \notin x''\) and since \(s'' \in \mathbb{S}^V\) we get the required result.

Case /PROC-\(\mathbb{S}^N\)/: Assume now that we have got a proof of

\((V, \text{env}) \vdash \langle \text{PROC } p(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 \text{ IN } c_2, s \rangle \rightarrow s'\)

from

(1) \((V, \text{env}(q = (x, y, c_1))) \vdash \langle c^{q}_2, s \rangle \rightarrow s'\)

where \(x' \notin V\).
where \( q \notin \text{DOM(env)} \) and furthermore assume that we have a proof of

\[(V, \text{env}) \vdash \langle \text{PROC } p(\text{VAL } x, \text{VAR } y) \rangle \text{IS } c_1 \text{ IN } c_2, s \rightarrow^{r'} s''.\]

This can only be obtained by applying the rule \(/\text{PROC-} q^N/\) to a formula of the form

\[(2) (V, \text{env}(q'=(x, y, c_1))) \vdash \langle c_{2p}, s \rangle \rightarrow^{r'} s''.\]

where \( q' \notin \text{DOM(env)} \). If \( q \equiv q' \) we can apply the induction hypothesis to (1) and (2) and get \( s' \equiv s'' \) and \( r = r' \) as required.

If \( q \not\equiv q' \) then we apply Lemma 4.1-6 to (2) and renames \( q' \) to \( q \). That is, we get

\[(V, \text{env}(q'=(x, y, c_1))) \vdash \langle c_{2p}, q', s \rangle \rightarrow^{r'} s''.\]

and thereby (since \( q' \notin \text{DOM(env)} \))

\[(V, \text{env}(q=(x, y, c_1))) \vdash \langle c_{2p}, s \rangle \rightarrow^{r'} s''.\]

Then the result follows using the induction hypothesis.

**Case /\text{CALL-} q^N/:** Assume that we have obtained a proof of

\[(V, \text{env}) \vdash \langle \text{CALL } p(e, y), s \rangle \rightarrow_{e(s) + e(s) + r}^{s'} s'',\]

from

\[(1) (V \cup \{ x \}, \text{env}) \vdash \langle c_{x', y', x} e(s), r \rangle \rightarrow^{s'} s',\]

where \( \text{env}(p) = (x', y', c) \) and \( x \) is a variable of the same sort as \( x' \) satisfying \( x \not\equiv \forall u \{ y \} \). We also have a proof of

\[(V, \text{env}) \vdash \langle \text{CALL } p(e, y), s \rangle \rightarrow^{r'} s'',\]

and it can only be obtained by applying \(/\text{CALL-} q^N/\) to a proof of

\[(2) (V \cup \{ x'' \}, \text{env}) \vdash \langle c_{x'' y''}, s_{x''} e(s), r'' \rangle \rightarrow^{s''} s''.\]

where \( x'' \not\equiv \forall u \{ y' \} \) and \( r'' = e(s) + e(s) + r'' \). If \( x'' \not\equiv x' \) then the required result follows directly from the induction hypothesis. If \( x'' \equiv x' \)
then we first rename $x''$ in (2) to $x'$ (very similar to the case /LET-$\mathcal{G}^N$/ above) and then apply the induction hypothesis. We omit the details as they are a straightforward modification of those for the case /LET-$\mathcal{G}^N$/ above.

Proof of Lemma 4.1-6

The proof is by induction on the length of the proof of $(V, env) \vdash \langle c_1, s \rangle \xrightarrow{L} s'$ in $\mathcal{N}$. Only two cases are interesting, namely those where either /PROC-$\mathcal{G}^N$/ or /CALL-$\mathcal{G}^N$/ is the last rule applied in the proof.

Case /PROC-$\mathcal{G}^N$/: Assume now that we have got a proof of

$$(V, env) \vdash \langle \text{PROC } p'(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 \text{ IN } c_2, s \rangle \xrightarrow{L} s'$$

from

$$(V, env(q'=(x, y, c_1))) \vdash \langle c_1 q'_{2p'}, s \rangle \xrightarrow{L} s'$$

where $q'$ is a procedure name satisfying $q' \notin \text{DOM}(env)$. We have six cases.

Case 1: $p \neq p'$ and $q' \notin q'$. The induction hypothesis can be applied to (1) since $q' \notin \text{DOM}(env)$ and we get

$$(V, env(q'=(x, y, c_1))) \vdash \langle c_2 q'_{2p'}, q'_{1p}, s \rangle \xrightarrow{L} s'$$

This means that

$$(V, env_p q'(=(x, y, c_{1p}))) \vdash \langle c_2', q', s \rangle \xrightarrow{L} s'$$

since $q' \notin \text{DOM}(env)$ and $p \notin \text{DOM}(env)$. Using /PROC-$\mathcal{G}^N$/ we get

$$(V, env_p q') \vdash \langle \text{PROC } p(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 q' \text{ IN } c_2, s \rangle \xrightarrow{L} s'$$

and thereby

$$(V, env_p q') \vdash \langle \text{PROC } p(\text{VAL } x, \text{VAR } y) \text{ IS } c_1 \text{ IN } c_2 q', s \rangle \xrightarrow{L} s'$$

335
Case 2: \(p \not\equiv p'\) and \(q \not\equiv q'\). Let \(q''\) be a procedure name satisfying \(q'' \notin \text{DOM}(env) \cup \{q'\}\). Then the induction hypothesis applied to (1) gives

\[
(V, env(q'=(x,y,c_1)))_p \vdash \langle c_{2p'}, q', s \rangle \sim_{s'}
\]

and thereby

\[
(V, env(q''=(x,y,c_1)))_p \vdash \langle c_{2p'}, s \rangle \sim_{s'}
\]

since \(q'' \notin \text{DOM}(env)\). The rest of the proof is now as in Case 1 above.

Case 3: \(p \not\equiv p', p \not\equiv q\) and \(q \not\equiv q'\). Since \(q \not\equiv \text{DOM}(env) \cup \{q'\}\) we can apply the induction hypothesis to (1) and get

\[
(V, env(q'=(x,y,c_1)))_p \vdash \langle c_{2p'}, q, s \rangle \sim_{s'}
\]

We then have

\[
(V, env(q'=(x,y,c_1)))_p \vdash \langle c_{2p'}, p, s \rangle \sim_{s'}
\]

since \(p \not\equiv q\). Using /PROC\^-qN/ we get

\[
(V, env)_p \vdash \langle \text{PROC} p'(VAL x, VAR y) IS c_{1p} IN c_{2p}, s \rangle \sim_{s'}
\]

and thereby

\[
(V, env)_p \vdash \langle \text{PROC} p'(VAL x, VAR y) IS c_{1} IN c_{2}, s \rangle \sim_{s'}
\]

as required.

Case 4: \(p \not\equiv p', p \not\equiv q\) and \(q \not\equiv q'\). We here use the same trick as in Case 2 above and rename \(q'\) to \(q''\) in (1) where \(q''\) is a procedure name satisfying \(q'' \notin \text{DOM}(env) \cup \{q\}\). Then we proceed as in Case 3 above. We omit the details.

Case 5: \(p \not\equiv p', p \not\equiv q\) and \(q \not\equiv q'\). First we apply the induction hypothesis to (1) and get

\[
(V, env(q'=(x,y,c_1)))_p \vdash \langle c_{2q}, q', s \rangle \sim_{s'}
\]
since \( q \in \text{DOM(env)} \). This means that

\[
(V, \text{env}_p q) = ((x, y, c_1 q' p') \mapsto c_2 q q' s) \rightarrow s',
\]

where \( q' \) is a procedure name satisfying \( q' \in \text{DOM(env)} \). Then \( /\text{PROC-}\phi^N/ \) gives

\[
(V, \text{env}_p q) \vdash (\text{PROC } q' (\text{VAL } x, \text{VAR } y) \text{ IS } c_1 q p \text{ IN } c_2 q' s) \rightarrow s',
\]

and thereby

\[
(V, \text{env}_p q) \vdash (\text{PROC } q (\text{VAL } x, \text{VAR } y) \text{ IS } c_1 q p \text{ IN } c_2 q' s) \rightarrow s',
\]
as required.

**Case 6:** \( p \neq p', p \neq q \) and \( q \neq q' \). We use the same trick as in Case 2 above and rename \( q' \) in (1). Then we proceed as in Case 5. We omit the details.

**Case /CALL-\phi^N/:** Assume now that we have got a proof of

\[
(V, \text{env} q') \vdash \text{CALL } p'(e, y),  s + e(s) + r \rightarrow s'
\]

from

\[
(1) \quad (V \cup x, \text{env}) q = c_{x, y, s} \rightarrow s',
\]

where \( \text{env}(p') = (x', y', c) \) and \( x \) is a variable of the same sort as \( x' \) satisfying \( x \notin \text{env}(y) \}. Using the induction hypothesis we get from (1) that

\[
(2) \quad (V \cup x, \text{env} q) q = c_{x, y, s} \rightarrow s'.
\]

We have two cases. If \( p \neq p' \) then \( \text{env}_p q = (x', y', c) \) so using \( /\text{CALL-}\phi^N/ \) we get from (2)

\[
(V, \text{env}_p q) \vdash \text{CALL } q(e, y), s + e(s) + r \rightarrow s',
\]

and thereby

\[
(V, \text{env}_p q) \vdash (\text{CALL } p(e, y)) q, s + e(s) + r \rightarrow s'.
\]
as required.

If \( p \neq p' \) then \( \text{env}_p^q(p') = (x', y', c_p^q) \) and \(/\text{CALL-}^N/\) applied to (2) gives

\[
(V, \text{env}_p^q) \vdash \text{CALL } p'(e, y), s \xrightarrow{s + e(s) + r} s'
\]

and thereby

\[
(V, \text{env}_p^q) \vdash (\text{CALL } p'(e, y))_{p'} s \xrightarrow{s + e(s) + r} s',
\]

as required. //
APPENDIX C: PROOFS OF PROPERTIES OF THE RECURSIVE
PROCEDURE LANGUAGE

The semantics and run-time requirements of the recursive procedure language is given by the formal system $\mathcal{R}$ of Section 5.1. In the sections 5.3 and 5.4 we use some properties holding for the language and we shall in this appendix sketch the proofs of some of them.

The first of these results allows one to extend the environment $\text{env}$ of a formula $(V,\text{env}) \vdash \langle c, s \rangle \xrightarrow{d} s'$:

Lemma 5.1-4: If $(V, \text{env}) \vdash \langle c, s \rangle \xrightarrow{d} s'$, $q \in \text{DOM}(\text{env})$, $F,V(c') \notin \cup \{x\}$ and $FP(c') \in \text{DOM}(\text{env}) \cup \{q\}$ then $(V, \text{env}(q=(x,c'))) \vdash \langle c, s \rangle \xrightarrow{d'} s'$ where $d' = d(q=0)$.

In the proof of this result we shall use the following lemma

Lemma 5.1-6: Assume that $(V,\text{env}) \vdash \langle c, s \rangle \xrightarrow{d} s'$, $p \in \text{DOM}(\text{env})$ and $q \in \text{DOM}(\text{env})$. Then $(V,\text{env}) \vdash \langle c, q \rangle \xrightarrow{d'} s'$ where $d'(p') = d(p')$ for $p' \in \text{DOM}(\text{env}) \setminus \{p\}$ and $d'(q) = d(p)$. Furthermore, the two proofs in $\mathcal{R}$ have the same lengths.

The proof of this result is a straightforward modification of that for the similar result in Section 4.1, Lemma 4.1-6 (the proof is given in Appendix B).

Proof of Lemma 5.1-4

The proof is by induction on the length of the proof of $(V,\text{env}) \vdash \langle c, s \rangle \xrightarrow{d} s'$ in $\mathcal{R}$. Only one case is interesting, namely that where \text{/PROC-}\$\mathcal{R}$/ is the last rule applied in the proof. The remaining cases are straightforward.
Case /PROC-/: Assume now that we have got a proof of

$$(V,\text{env}) \vdash \langle \text{PROC } p(\text{VAL } x') \rangle \text{ IS } c_1 \text{ IN } c_2, s \xrightarrow{d} s'$$

by applying the rule /PROC-$/R/ to a proof of

$$(1) \ (V,\text{env}(p'=(x',c_{1p'}))) \vdash \langle c_{2p'}, s \xrightarrow{d'} s'$$

where $p'$ is a procedure name satisfying $p' \notin \text{DOM(env)}$ and $d'(p'')=d(p'')$ for $p''$ in $\text{DOM(env)}$. We can then apply the induction hypothesis to (1) in the case where $p' \notin q$ and get

$$(V,\text{env}(p'=(x',c_{1p'}))(q=(x',c'))) \vdash \langle c_{2p'}, s \xrightarrow{d'} s'$$

and using /PROC-$/R/ we get the required proof of

$$(V,\text{env}(q=(x,c')))) \vdash \langle \text{PROC } p(\text{VAL } x') \rangle \text{ IS } c_1 \text{ IN } c_2, s \xrightarrow{d''} s'$$

where $d''(p'')=(d'(q=0))p''$ for $p'' \notin \text{DOM(env)} \cup \{q\}$. But then $d''=d(q=0)$ as required.

If $p' \notin q$ we shall rename $p'$ to $q'$ where $q' \notin \text{DOM(env)} \cup \{q\}$. Lemma 5.1-6 applied to (1) gives

$$(V,\text{env}(p'=(x',c_{1p'}))q') \vdash \langle c_{2p'}, s \xrightarrow{d'} s'$$

where $d'_1(p'')=d'(p'')$ for $p'' \notin \text{DOM(env)} \cup \{p'\}$ and $d'_1(q')=d'(p')$. Since $p' \notin \text{DOM(env)}$ this means that

$$(2) \ (V,\text{env}(q'=(x',c_{1p'}))) \vdash \langle c_{2p'}, s \xrightarrow{d'} s'$$

The induction hypothesis can now be applied to (2) since the proofs of (1) and (2) have the same lengths. The rest of the proof is as in the case $p' \notin q$ above so we omit the details.

Furthermore, we shall sketch the proofs of the following results allowing us to replace one formal parameter with another one in certain situations.
Lemma 5.1-7: Assume that \( a \in \text{V-FV}(\text{env} \mid \text{FP}(\text{CALL p(a),env})) \) and \( \text{FV}(e) \subseteq \text{V} \).

If \((V, \text{env}) \Downarrow \langle \text{CALL p(a), s} \rangle \Downarrow_d s'\) then \( r = a^e(s \mid a)(s \mid a) + \tau'\)

for some \( \tau' \) and \((V, \text{env}) \Downarrow \langle \text{CALL p(e), s} \rangle \Downarrow_d s''\) for some state \( s'' \) satisfying \( s'' \models \text{V} \langle a \rangle s' \).

Lemma 5.1-8: Assume that \( a \in \text{V-FV}(\text{env} \mid \text{FP}(\text{CALL p(e),env})) \).

If \((V, \text{env}) \Downarrow \langle \text{CALL p(e), s} \rangle \Downarrow_d s'\) then \( r = e^e(s \mid e)(s \mid e) + \tau'\)

for some \( \tau' \) and \((V, \text{env}) \Downarrow \langle \text{CALL p(a), s} \rangle \Downarrow_d s''\) for some state \( s'' \) satisfying \( s'' \models \text{V} \langle a \rangle s' \).

The proof of the first of these results is as follows:

Proof of Lemma 5.1-7

Assume that \( \text{env}(p') = (x', c) \). From

\[(V, \text{env}) \Downarrow \langle \text{CALL p(a), s} \rangle \Downarrow_d s'\]

and \( /\text{CALL}-^a/ \) we get that \( r = a^e(s \mid a)(s \mid a) + \tau'\)

for some \( \tau' \) and \((V \cup x, \text{env}) \Downarrow \langle \text{CALL p(e), s} \rangle \Downarrow_d s'\)

where \( s_0 \) abbreviates \( a \in \text{env}(x)' \), \( x \) is a variable of the same sort as \( x' \)

satisfying \( x \notin \text{V} \) and \( d' \) is the counter \( d(p = d(p) - 1) \) (note \( d(p) > 0 \) must hold). We have \( a(s_0) = e(s) \) so \( s_0 \models (V \cup x) \langle a \rangle s_x \). From the assumption \( a \in \text{V-FV}(\text{env} \mid \text{FP}(\text{CALL p(a),env})) \) and \( x \notin \text{V} \) we get

\( a \in \text{P-FV}(x) \cup \text{FP}(x, \text{env})) \) so Lemma 5.1-2 gives

\((V \cup x, \text{env}) \Downarrow \langle \text{CALL p(a), s} \rangle \Downarrow_d s'\)

for some \( s' \) satisfying \( s' \models (V \cup x) \langle a \rangle s' \). Using Lemma 5.1-3 we then get

\((V \cup x, \text{env}) \Downarrow \langle \text{CALL p(e), s} \rangle \Downarrow_d s'\)

for some \( s' \) satisfying \( s' \models (V \cup x) \langle a \rangle s' \). So the semantic rule \( /\text{CALL}-^a/ \)
\[
(V, \text{env}) \vdash \langle \text{CALL } p(e), s \rangle \xrightarrow{e(s) + e(s) + r} d, s' \]

Since \( s'^2 \vdash \{ x \} \) holds we have completed the proof of the lemma. ///

**Proof of Lemma 5.1-8**

A proof of this result can be obtained by essentially reading the proof above backwards so we omit the details. ///

In Section 5.4 we need a couple of results about transformed versions of the programs. As for the definition of the notation we refer to Section 5.4. First we have

**Lemma 5.4-1:** If \((V_1, \text{env}) \vdash \langle c', s \rangle \xrightarrow{d} s'\) for some program \( c' \) and environment \( \text{env} \) with \( z \in \text{FV}(c') \cup \text{FV}(\text{env}) \) and where \( z(s).d(p) \) then for some \( s'' \) and \( r' \) \((V_1, \text{env}(p=(x, c'))) \vdash \langle c', s'' \rangle \xrightarrow{r'} s''\) and \( s'' \vdash \{ z \} \).

\( z(s'') = z(p) \) and the two proofs in \( \mathcal{F}_R \) have the same lengths. ///

**Proof:** The lemma can be proved by induction on the length of proofs in \( \mathcal{F}_R \). The cases where the last axiom or rule applied in the proof is one of \(/\text{ass-}\mathcal{F}_R/\), \(/\text{IF-}\mathcal{F}_R/\), \(/\text{WHILE-}\mathcal{F}_R/\), \(/\text{LET-}\mathcal{F}_R/\) and \(/\text{PROC-}\mathcal{F}_R/\) are straightforward so we omit the details. Only one case is left:

**Case */CALL-\mathcal{F}_R/*:** Assume that we have a proof of

\[
(V_1, \text{env}) \vdash \langle \text{CALL } p'(e), s \rangle \xrightarrow{e(s) + e(s) + r} d(p=d(p)+1) s',
\]

from

\[
(1) \quad (V_1, \text{env}(y), \text{env}) \vdash \langle c', y \xrightarrow{e(s)} s \rangle \xrightarrow{r} d s',
\]

where \( \text{env}(p')=(x', c') \) and \( y' \) \((V_1)\) has the same sort as \( x' \). If \( p \not= p' \) we can apply the induction hypothesis to (1) and then the required result follows using the rule */CALL-\mathcal{F}_R/*.

342
So assume that $p \neq p'$. By assumption we have $z(p) \geq (d(p) + 1)(p)$, so $z(p) \geq d(p) + 1$. Since $z(p)^{\text{FV}}(c) = \text{FV}(\text{env})$ and $z(p)^{\text{FV}}(s) \geq z(p) - 1$, we get from Lemma 5.1-2 that

$$
((V_{\lambda} \{ y \}, \text{env}) \triangleright \langle c_{x, y}^p, s_{x, y}^p \rangle_{z_{p}}) \rightarrow d_{s_1}^{s_1'}
$$

where $s_1^\prime = (V_{\lambda} \{ y \}, \text{env}) \triangleleft z_{p}^p s_1$. Lemma 5.1-3 then gives

$$
((V_{\lambda} \{ y \}, \text{env}) \triangleright \langle c_{x, y}^p, s_{x, y}^p \rangle_{z_{p}}) \rightarrow d_{s_2}^{s_2'}
$$

where $s_2^\prime = (V_{\lambda} \{ y \}, \text{env}) \triangleleft z_{p}^p s_2'$. Since both Lemma 5.1-2 and Lemma 5.1-3 preserve the length of proofs in $\text{FV}$ and $e(s) z(p) - 1$, we can apply the induction hypothesis and get

$$
((V_{\lambda} \{ y \}, \text{env}(p = (x, c_p))) \triangleright \langle c_{x, y}^p, s_{x, y}^p \rangle_{z_{p}}) \rightarrow d_{s_3}^{s_3'}
$$

for some $s_3' = (V_{\lambda} \{ y \}, \text{env}(p = (x, c_p))) \triangleleft z_{p}^p s_3'$ and $z(p) s_3' = z(p) - 1$. It can be proved that

$$
((V_{\lambda} \{ y \}, \text{env}(p = (x, c_p))) \triangleright \langle c_{x, y}^p, s_{x, y}^p \rangle_{z_{p}}) \rightarrow d_{s_0}^{s_0'}
$$

where $d_0(p) = 0$ for $p$ in $\text{DOM}(\text{env})$ and $s_0^\prime = (V_{\lambda} \{ y \}, \text{env}(p = (x, c_p))) \triangleleft z_{p}^p s_0$. Furthermore, Lemma 5.1-2 gives

$$
((V_{\lambda} \{ y \}, \text{env}(p = (x, c_p))) \triangleright \langle c_{x, y}^p, s_{x, y}^p \rangle_{z_{p}}) \rightarrow d_{s_4}^{s_4'}
$$

for $s_4^\prime = (V_{\lambda} \{ y \}, \text{env}(p = (x, c_p))) \triangleleft z_{p}^p s_4'$. We have

$$
(V_{\lambda} \{ y \}, \text{env}(p = (x, c_p))) \triangleright \langle c_{x, y}^p, s_{x, y}^p \rangle_{z_{p}} \rightarrow d_{s_4'^{s_4'+1}}^{s_4'^{s_4'+1}}
$$

The semantic rules $/\text{IF-}\text{FV}/$ and $/\text{IF-}\text{FV}/$ together with the fact that $z(p) s_4' = z(p) - 1$ give

$$
(V_{\lambda} \{ y \}, \text{env}(p = (x, c_p))) \triangleright \langle c_{x, y}^p, s_{x, y}^p \rangle_{z_{p}} \rightarrow d_{s_4'^{z(p)}}^{s_4'^{z(p)}}
$$

The semantic rule $/\text{CALL-}\text{FV}/$ now gives

$$
(V_{\lambda} \{ y \}, \text{env}(p = (x, c_p))) \triangleright \langle \text{CALL p(e), s} \rangle_{d(p + 1)} \rightarrow d_{s_4'^{z(p)}}^{s_4'^{z(p)}}
$$

It is easy to verify that $s_4'^{z(p)} \geq s_1'$ and the result follows.
Finally, we shall prove the following result:

**Lemma 5.4-3:** If \((V_1, \text{env}_1) \vdash <c, s> \overset{L_2}{\rightarrow} s'\) for some program with \(\text{FV}(c) \leq c\)

\(V_1 - \text{VAR}(L_1)\) and \(\text{FP}(c) \cap \text{PROC}(L_1)\) then \(s \overset{L_1}{\rightarrow} d, d(p') = 0\) for \(p' \in \text{PROC}(L_1)\)

and \((V_1, \text{env}) \vdash <c, s> \overset{L_2}{\rightarrow} s''\) for some \(s''\) and \(r'\) where \(s'' \overset{L_1 - \text{VAR}(L_1)}{\rightarrow} s'\).

**Proof:** The lemma can be proved by induction on the length of proofs in \(\mathcal{G}^R\). The cases where the last axiom or rule applied in the proof is one of \(/\text{ass}-\mathcal{G}^R/, /\text{IF}-\mathcal{G}^R/, /;_{-}\mathcal{G}^R/, /\text{WHILE}-\mathcal{G}^R/\) and \(/\text{LET}-\mathcal{G}^R/\) are straightforward so we omit the details. The remaining two cases are handled as follows:

**Case /PROC-\mathcal{G}^R/:** Assume that

\[(V_1, \text{env}_1) \vdash \text{PROC} p(\text{VAL} x) \quad \text{IS} \quad c_1 \quad \text{IN} \quad c_2, s \overset{L_2}{\rightarrow} s'\]

is obtained by applying \(/\text{PROC}-\mathcal{G}^R/\) to a proof of

\[(1) \quad (V_1, \text{env}_1(q = (x, c_1 q))) \vdash <c_2q, s_q \overset{r_q}{\rightarrow} d_q, s_q'\]

where \(q \in \text{DOM}(\text{env}_1)\) and \(d' = d(q = n)\) for some \(n\). If \(p \notin \text{FP}(c_2)\) we can directly apply the induction hypothesis. Since \(z \notin \text{env}_1\) we can apply Lemma 5.1-2 and Lemma 5.1-3 to (1) and get

\[(V_1 \cup \{z\}, \text{env}_1(q = (x, c_1 p))) \vdash <c_2q, s_q^n \overset{r_q}{\rightarrow} d_q, s_q'\]

for \(s_q \overset{1}{\rightarrow} s_q'\) and \(z_q(s_q') = n\). Since \(z \notin \text{FP}(c_2)\) we can directly apply the induction hypothesis and get

\[(V_1 \cup \{z\}, \text{env}_1(q = (x, c_1 q))) \vdash <c_2q, s_q^n \overset{r_q}{\rightarrow} d_q, s_q'\]

for \(s_q \overset{1}{\rightarrow} s_q'\). We have \(\text{env}_1(q = (x, c_1 p)) \equiv (\text{env}(q = (x, c_1 q)))_q\). Since the transformation on proofs performed by the lemmas 5.1-2 and 5.1-3 and 5.4-1 do not change their length we can apply the induction hypothesis and get \(s_q^n \overset{L_1 \cup \{q, z\}}{\rightarrow} d', d(p') = 0\) for \(p' \in \text{PROC}(L_1)\) and
(V_1 u \{z_q\}, \text{env}) \vdash \text{VAR}(L_1) u \{z_q\} s_2'.
Using first /PROC-\Sigma^R/ and then Lemma 5.1-2 to remove \( z \) we get

\( (V_1, \text{env}) \vdash \text{PROC } p(\text{VAL } x) \text{ IS } c_1 \text{ IN } c_2, s \to d', s_4' \)

where \( s_4' \sim s_3' \). It is easy to verify that \( s_4' \sim V_1 \text{-VAR}(L_1) s', s_2 \sim L_1 d \)
and \( d(p')=0 \) for \( p' \in \text{PROC}(L_1) \).

**Case /CALL-\Sigma^R/:** Assume now that we have got a proof of

\( (V_1 u \{y\}, \text{env}) \vdash \text{CALL } p(e), s \to e(s) + e(s) + r ) \frac{e(s)^+}{d(p=d(p)+1)} s' \to d', s_4' \)

from

1. \( (V_1 u \{y\}, \text{env}) \vdash \text{CALL } p(e), s \to e(s) + e(s) + r ) \frac{e(s)^+}{d(p=d(p)+1)} s' \to d', s_4' \)

where \( \text{env}(p)=(x,c) \) and \( y (\notin V_1) \) has the same sort as \( x \). We must have

\( z_p(s)>0 \) since otherwise the program \( c \in p x \) would loop. So there will be states \( s_1 \) and \( s_2 \) such that

2. \( (V_1 u \{y\}, \text{env}) \vdash \text{CALL } p(e), s \to e(s) + e(s) + r ) \frac{e(s)^+}{d(p=d(p)+1)} s' \to d', s_4' \)

and

3. \( (V_1 u \{y\}, \text{env}) \vdash \text{CALL } p(e), s \to e(s) + e(s) + r ) \frac{e(s)^+}{d(p=d(p)+1)} s' \to d', s_4' \)

for some \( r_1, r_2, r_3, d_1, d_2 \) and \( d_3 \). This follows from the semantic rules

/IF-\Sigma^R/ and /;\Sigma^R/. From (2) it can be proved that \( s_1 \sim V_1 u \{y\} \)

\( e(s), z_p(s)=1 \) \( \text{and } d_1(p')=0 \) for \( p' \in \text{DOM}(\text{env}) \) (using Lemma 5.1-9 stating

that the language is deterministic). Similarly we get from (4) that

\( z_p(s)=1 \)

\( s_2=s_1' \)

\( d_1(p')=0 \) for \( p' \in \text{DOM}(\text{env}) \). Using Lemma 5.1-2 we

get from (3) that

\( (V_1 u \{y\}, \text{env}) \vdash \text{CALL } p(e), s \to e(s) + e(s) + r ) \frac{e(s)^+}{d(p=d(p)+1)} s' \to d', s_4' \)

where \( s_2 \sim V_1 u \{y\} \)

\( z_p(s)=1 \)

and using that \( d=d_3 \) must hold. The induction hypothesis can now be applied and we get

\( s_2 \sim V_1 u \{y\} \)

\( z_p(s)=1 \)

\( \text{and } d=1 \)
d(p') = 0 for p' \not\in \text{PROC}(L_1) \text{ and } e(s) = z_p(s) - 1

(V_1, \{y\}, env) \triangleright \text{CALL } y \stackrel{\chi}{\rightarrow} x \triangleright \gamma \frac{r}{z_p} s^3

where s^3 = (V_1 \cup \{y\}) - \text{VAR}(L_1) s^3_1. \text{ Using first the semantic rule } /\text{CALL-}\gamma/\text{ then Lemma 5.1-2 with } s^3_p(s) - 1, \text{z}_{V_1 - \{z\}} s \text{ and finally Lemma 5.1-3 (to add } z_p \text{ again) we get }

(V_1, env) \triangleright \text{CALL } p(e), s \triangleright \frac{e(s) + e(s) + r}{2} s^4

where s^4_3 = (V_1 - \{z\}) s^3_1. \text{ It is straightforward to verify that } s^4_3 \geq d(p = d(p) + 1) \text{ and } s^4_1 = V_1 - \text{VAR}(L_1) s_1'. \text{ This completes the proof of the lemma.} \text{ } //
REFERENCES

/Ac82/ P. Aczel: A note on program verification, Manchester University, 1982


/AlAr78/ S. Alagić and M.A. Arbib: The Design of Well-structured and Correct Programs, Springer Verlag, 1978

/AsTu82/ P.R.J. Asveld and J.V. Tucker: Complexity theory and the operational structure of algebraic programming systems, Acta Inf. 17, 451-476, 1982

/BeTu80/ J.A. Bergstra and J.V. Tucker: Expressiveness and the completeness of Hoare's logic, Rep IW 149/80, Mathematisch Centrum, Amsterdam, 1980


E.M. Clarke: Programming language constructs for which it is impossible to obtain good Hoare axiom systems, J.ACM 26, 129-147, 1979

S.A. Cook: Soundness and completeness of an axiom system for program verification, SIAM J. Comput. 7, 70-90, 1978


G.A. Gorelick: A complete axiomatic system for proving assertions about recursive and non-recursive programs, MSc Thesis, University of Toronto, 1975


D. Harel, A. Pnueli and J. Stavi: A complete axiomatic system for proving deductions about recursive programs, in Proc. of the Ninth Symp. on Theory of Comp., 249-260, 1977
/Ha79/ D. Harel: First Order Dynamic Logic, LNCS 68, Springer Verlag, 1979


/HoU179/ J. E. Hopcroft and J. D. Ullman: Introduction to Automata Theory, Languages and Computation, Addison Wesley, 1979


/KaAd80/ T. Kasai and A. Adachi: A characterization of time complexity by simple loop programs, JCSS 20, 1-17, 1980


/Ko83/ D. Kozen: A probabilistic PDL, in Proc of the Fifteenth Symp. on Theory of Comp., 1983

/Li77/ R. J. Lipton: A necessary and sufficient condition for the existence of Hoare logics, in Proc 18th IEEE Symp. Found. of Comp. Sci., 1-6, 1977

/LiZi75/ B. Liskov and S. Zilles: Specification techniques for data abstraction, IEEE Transactions for Software Engineering 1, 7-19, 1975


/LuSu77/ D. C. Luckham and N. Suzuki: Proof of termination within a weak logic of programs, Acta Inf. 8, 21-36, 1977
/MaPn74/ Z. Manna and A. Pnueli: Axiomatic approach to total correctness of programs, Acta Inf. 3, 253-263, 1974


/Na66/ P. Naur: Proof of algorithms by general snapshots, BIT 6, 310-316, 1966

/Ni83/ H.R. Nielson: Proof systems for computation time, Third Conf. on Software Technology and Theoretical Computer Science, 1983

/Ol80/ E.R. Olderog: General equivalence of expressivity definitions using strongest post-conditions resp. weakest pre-conditions, Ber. 8007 University of Kiel, 1980


/Sh67/ J.R. Shoenfield: Mathematical Logic, Addison Wesley, 1967


/So77a/ S. Sokolowski: Axioms for total correctness, Acta Inf., 61-72, 1977


/St77/ J.E. Stoy: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT-Press, 1977


/Wa78/ M.Wand: A new incompleteness result for Hoare's system, J.ACM 25, 168-175, 1978

/We76/ B.Wegbreit: Verifying program performance, J.ACM 23, 691-699, 1976

/Wi71/ N.Wirth: The programming language Pascal, Acta Inf. 1, 35-63, 1971
INDEX

arithmetical completeness, 70
arithmetical computational model, 79
arithmetical model, 69
arithmetical sound, 70
assertion language, 47

boolean expression, 42
book-keeping trick, 308, 312
bubble sorting, 109, 276
closure, 145, 200
completeness, 9, 13
Completeness Theorem
for 3, 131
for 5, 126, 317
for 2, 100
for 3N, 165
for 3R, 229
for 1, 61
for 4, 281, 292
cost-function, 78
counter, 201

data type, 39
containing Peano Arithmetic, 69
    with minus operator, 291
depth counter, 201
Dijkstra's graph algorithm, 301
direct style approach, 27

equivalence problem, 174
exact time complexity, 15
exact time term expressiveness condition, 118
expressiveness condition, 13
    for 3, 125
    for 2, 99
    for 3N, 164
    for 3R, 228
    for 1, 60
    for 4, 281, 290
Extended Peano Arithmetic, 81

factorial example, 48, 53, 92, 213, 256
fibonacci example, 158
free procedure names, 145
free variables, 42, 144
function symbols, 39

informal rules for run-time analysis, 17
initial state - final state relation, 43
K-sorted signature, 39

loop programs, 17, 140
method of programmed counters, 19, 118, 120
merge sorting, 33, 259
model, 40

natural formalisation, 9
numerical computational model, 79
    with subtraction, 291
numerical model, 55
one-dimensional arrays, 81
order of magnitude, 15

partial correctness, 14
Peano Arithmetic, 39, 41, 79

352
post-condition, 10
pre-condition, 10
procedure name, 144
procedure program, 144
program variable, 41
proof from assumption, 197
pure formula, 48
reasonable environment, 145
relational formula, 48
relation symbol, 39
shadow variable, 47
Shaw's method, 20, 122
sort, 39
soundness, 8, 13
Soundness Theorem
   for \( \mathfrak{B} \), 131
   for \( \mathfrak{T} \), 125
   for \( \mathfrak{A} \), 95
   for \( \mathfrak{A}^N \), 160
   for \( \mathfrak{A}^R \), 216
   for \( \mathfrak{T} \), 56
   for \( \mathfrak{U} \), 280
standard model
   of Extended Peano Arithmetic, 81
   of Peano Arithmetic, 41
state, 43
strongest post-condition, 13
strong expressiveness condition
   for \( \mathfrak{U} \), 290
term, 41
time expressiveness condition, 86
time formula, 85
time term, 272
time term expressiveness condition,
   272
total correctness, 14

trace list, 232
uniform computational model
   for Extended Peano Arithmetic, 81
   for Peano Arithmetic, 79
uniform cost-criteria, 79
union-find problem, 174
unknown time formulas, 208
validity, 50, 89, 123, 128, 155, 208, 274
d: L-validity, 208
well-formed formula
   of \( \mathfrak{T} \), 123
   of \( \mathfrak{A} \), 89
   of \( \mathfrak{A}^N \), 155
   of \( \mathfrak{A}^R \), 207
   of \( \mathfrak{T} \), 50
   of \( \mathfrak{U} \), 274
well-formed program, 145
well-formed relative to L, 207
while program, 42
worst-case time complexity, 15
b, 42
\nu, 45, 148
\nu, 49
Q, 129
T, 123, 124
H, 11
m, 40
m_k, 40
n, 41
\nu, 90
X^N, 156
X^R, 209
\varepsilon, 84
\rho, 44
\rho_a, 151
\rho_R, 202
T, 46, 51
U, 275
p, 144
\rho, 42
p^+, 79
P, 48
\overline{P}, 49
q, 144
Q, 48
r, 83
R, 85
s, 43
time, 85
T, 272
x, 42
\overline{x}, 47
X, 49
X', 49
\overline{X}, 49
X^q, 147
\nu^v, 153
X^q, 149
\nu^q, 154
E^+(x), 86
E_\ell(x_1, \ldots, x_m), 86
E(x_1, \ldots, x_m), 87
U(x), 118, 272
U_\ell(x_1, \ldots, x_m), 118, 272
U_p(x_1, \ldots, x_m), 118, 273
E(b), 88
E(e), 88
E^+(e), 88
E^S(e), 88
U(b, 119, 273) U(e, 119, 273) U(S(e), 119, 274) G(c), 60, 99, 290 E(c), 99 WPe(c), R, 125 G(v, env)(c), 164

TRUE, 42 FALSE, 42
x := e, 42, 144, 200
IF b THEN c_1 ELSE c_2, 42, 144, 200
c_1 ; c_2, 42, 144, 200
WHILE b DO c, 42, 144, 200
IF b THEN c, 140
SKIP, 140
REPEAT c UNTIL b, 141
FOR x = e_1 TO e_2 DO c, 141
LET x = e IN c, 144, 200
PROC p(VAL x, VAR y) IS c_1 IN c_2, 144
CALL p(e, y), 144
PROC p(VAL x) IS c_1 IN c_2, 200
CALL p(e), 200
P[c]Q, 10, 197
P(c)Q/V, 46
P(c)RQ/V, 89
P&R(c)Q&R'/V, 122, 127
P(c)RQ/(V, env), 154
P[c_1:R_1]Q_1 & & P[c_k:R_k]Q_k
  => P_0[c_0:R_0]Q_0/(V, env, L), 206
P(c)TQ/V, 272
<table>
<thead>
<tr>
<th>Example 2.1-1:39</th>
<th>Lemma 3.1-1:85</th>
<th>Lemma 5.1-6:205,339</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 2.2-1:48</td>
<td>- 3.2-1:88</td>
<td>- 5.1-9:206</td>
</tr>
<tr>
<td>- 3.1-1:79</td>
<td>- 3.4-1:107</td>
<td>- 5.3-1:216</td>
</tr>
<tr>
<td>- 3.1-2:80</td>
<td>- 3.6-1:119</td>
<td>- 5.3-2:216</td>
</tr>
<tr>
<td>- 3.1-3:81</td>
<td>- 4.1-1:152</td>
<td>- 5.3-3:217</td>
</tr>
<tr>
<td>- 3.2-1:87</td>
<td>- 4.1-2:152</td>
<td>- 5.3-4:217</td>
</tr>
<tr>
<td>- 3.4-1:108</td>
<td>- 4.1-3:152,325</td>
<td>- 5.4-1:235,342</td>
</tr>
<tr>
<td>- 5.4-1:133</td>
<td>- 4.1-4:153,325</td>
<td>- 5.4-2:235</td>
</tr>
<tr>
<td>- 6.1-1:273</td>
<td>- 4.1-5:153,325</td>
<td>- 5.4-3:235,344</td>
</tr>
<tr>
<td>- 6.2-1:281</td>
<td>- 4.1-6:154,326</td>
<td>- 5.4-4:236</td>
</tr>
<tr>
<td></td>
<td>- 5.1-1:203</td>
<td>- 5.4-5:236</td>
</tr>
<tr>
<td>Lemma 2.1-1:45</td>
<td>- 5.1-2:203</td>
<td>- 6.1-1:274</td>
</tr>
<tr>
<td>- 2.1-3:46</td>
<td>- 5.1-4:204,339</td>
<td>- 6.2-1:282</td>
</tr>
<tr>
<td>- 2.4-1:70</td>
<td>- 5.1-5:204</td>
<td>- 6.2-2:283</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 6.3-1:298</td>
</tr>
</tbody>
</table>