Formal definitions of programming languages as a basis for compiler construction

FORMAL DEFINITIONS OF PROGRAMMING LANGUAGES AS A BASIS FOR COMPILER CONSTRUCTION

C. HEMERIK

FORMAL DEFINITIONS OF PROGRAMMING LANGUAGES AS A BASIS FOR COMPILER CONSTRUCTION

Druk: Dissertatie Drukkerij Wibro, Helmond, Telefoon 04920-23981.

FORMAL DEFINITIONS OF PROGRAMMING LANGUAGES AS A BASIS FOR COMPILER CONSTRUCTION

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE TECHNISCHE WETENSCHAPPEN AAN DE TECHNISCHE HOGESCHOOL EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.DR. S.T.M. ACKERMANS, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENBAAR TE VERDEDIGEN OP DINSDAG 15 MEI 1984 TE 16.00 UUR

DOOR

CORNELIS HEMERIK

GEBOREN TE LEIDEN

Dit proefschrift is goedgekeurd
door de promotoren

prof.dr. F.E.J. Kruseman Aretz

en

prof.dr. E.W. Dijkstra

CONTENTS

CHAPTER 0
INTRODUCTION

0.1. Background

In order to place the subject of this thesis in the proper perspective we shall first devote a few words to the research project of which it is a part. The aim of the latter project is the systematic construction of correct compilers based on formal definitions of both source and target language. Let us make this more precise:

If we want to construct a compiler from a source language SL to a target language TL we have to take into account at least the following aspects:

1. The definition of SL.
2. The definition of TL.
3. The construction of a "meaning preserving" mapping from SL to TL.
4. The construction of a program that realizes that mapping.

To a mathematically inclined person the dependencies between these aspects are obvious: 3 depends on 1 and 2, and the specifications used in 4 are based on 3. It is also clear that the correctness concerns of 3 and 4 can be separated and that the reliability of the resulting compiler ultimately depends on the rigour of 1 and 2. In practice, mainly due to historical causes, the situation is different however:

- Compiler construction is a relatively old branch of computing science, whereas the mathematical theory of programming and programming languages has not matured until the last decade. Consequently the formalization of many programming concepts has lagged far behind their implementation. To implementers (and many others) the operational view still prevails and formal definitions have been considered, in the terminology of [Ashcroft], descriptive rather than prescriptive.
- The few research efforts in compiler correctness have concentrated on formal models of translators that have been used in elaborate proofs of completely trivial language mappings. Attention has been paid to

correctness proofs of given mappings rather than to the construction of correct mappings. Moreover, the connection between such an abstract mapping and a concrete compiler has not always been clear.

- At present compiler construction often proceeds by the construction of a parser which is subsequently augmented with various symbol table manipulation and code generation routines. Thus the language mapping realized by such a compiler is only specified implicitly. Explicit compiler specifications are rare and as a consequence the programming discipline where program and correctness proof are developed hand in hand is seldom applied to compilers.

We are convinced that at present formal language theory and programming methodology have developed sufficiently to make an intellectually more satisfying approach to compiler construction feasible. To turn that conviction into fact we have set as our goal the construction of a compiler along the lines of points 1-4 above. More specifically, this includes the following tasks:

- Design and formal definition of a source language SL and a target language TL. This task involves the development of formal definition methods to the extent that languages can be defined completely, i.e. that both language-theoretical results, implementations, and programmer-oriented aspects such as proof rules may be derived from the formal definition.
- The systematic derivation of a mapping from SL to TL. This task involves the development of some theory concerning correctness of translations as well as application of that theory to the problem at hand.
- Specification of a compiler based on the derived mapping, followed by construction of a program conforming to that specification.

It has turned out that the main difficulties are in the first task. It is this task that is the subject of the thesis. In section 0.2 we shall describe it in more detail. The remainder of the project will be described in a subsequent report.

0.2. Subject of the thesis

As already mentioned, the subject of this thesis is the design and formal definition of a source language SL and a target language TL, together with the development of supporting definition methods. Our aim is to obtain language definitions which present programs as mathematical objects free of reference or commitment to particular implementations, but which are also sufficiently complete and precise to derive correct implementations from. From the background sketched in section 0.1 it will be clear that this thesis should not be considered as an isolated and self-contained study on formal language definition. The major part of the work reported here is intended as theoretical foundation of the aforementioned work on compiler correctness. We emphasize this background because it may not be obvious from the outer appearance of this thesis, although it is of significant influence on its subject matter, e.g. in the following respects:

- This thesis is concerned neither with development of general definition methods, nor with general theory concerning such methods. Rather it is concerned with development of formal tools which are both theoretically well-founded and practically usable. The mathematical apparatus needed for this purpose is only developed as far as necessary.
- Most work on formal definition of programming languages is concerned with either syntax or semantics; in order to obtain compiler specifications we have to consider both. We also pay much attention to context-dependent syntax, a subject which is usually considered semantic in studies on syntactic analysis and syntactic in studies on semantics. Context-dependent syntax plays an important role in compiler construction, but also affects the semantics of constructs involving changes of context, such as blocks and procedures.
- In chapter 5 we develop predicate transformer semantics [Dijkstra 1, Dijkstra 2] for typical machine language sequencing primitives such as jumps. We do so not to liberate these constructs from their "harmful" reputation, but to facilitate the derivation of mappings from SL- to TL-programs from correspondences between their semantics.

We hope to have made clear in what light this thesis should be seen. We continue with an overview of its contents:

In chapter 1 we consider the role of formal definitions of programming languages, we formulate some principles and criteria regarding their use, and we motivate the form and design choices of the definitions in subsequent chapters.

In chapter 2 we investigate how the principles of chapter 1 can be applied to the definition of the syntax of the source language. The main subject of the chapter is the development of a variant of the well-known attribute grammars [Knuth] which is primarily aimed at language specification. The main components of this variant are a collection of parameterized production rules and a so-called attribute structure by means of which properties of parameters can be derived from given axioms. On the one hand an attribute grammar of this kind may be viewed as a self-contained formal system based on rewrite rules and logical derivations. On the other hand the attribute structure, which corresponds to an algebraic data type specification in the sense of [Goguen, Guttag], can be used directly as specification of the context-dependent analysis part of a compiler.

In chapter 3 we lay the basis for the semantic definitions of both source and target language. The semantic definition method we employ is essentially that of Dijkstra's predicate transformers [Dijkstra 1, Dijkstra 2]. First we provide a foundation for this method by means of a variant of Scott's lattice theory [Scott 2] and infinitary logic [Back 1, Karp]. Subsequently we study predicate transformers for the kernel language in this lattice-theoretical framework. Finally we use these results to develop partial and total correctness logics in the style of [Hoare 1, Hoare 2], and we prove soundness of these logics with respect to predicate transformer definitions.

In chapter 4 the application of the methods of chapters 2 and 3 is extended to other constructs of the source language, viz. blocks and procedures, for which both syntax, semantics, and proof rules are developed. The various aspects of procedures are considered in isolation as much as possible. In section 4.1 we discuss blocks to investigate the effects of the introduction of local names. Section 4.2 deals with so-called abstractions which are used to study the effects of parameterization. Section 4.3 concentrates on recursion, which can be handled rather easily by means of the lattice theory of section 3.1. Finally, in section 4.4 the various aspects are merged, resulting in a treatment of parameterized recursive procedures.

In chapter 5 we consider some aspects of the formal definition of the target language TL, viz. those that have to do with sequencing. The main goal of this work is to obtain predicate transformer semantics for machine instructions, which can be used in compiler correctness arguments. First we develop predicate transformer semantics based on the lattice theory of section 3.1 and the continuation technique of denotational semantics [Strachey]. Thereafter we derive an equivalent operational description by means of an interpreter. This derivation can be considered both as a consistency proof of two definitions and as a derivation of an implementation from a non-operational definition. In addition, it also gives an impression of the semantics preserving transformations that will be used in the translation from source language to target language.

Chapter 6 contains some concluding remarks.
Appendix A contains proofs of some lemmas.
Appendix B contains the collected definitions of the source language.

0.3. Some notational conventions

- Definitions and theorems may consist of several clauses, and are numbered sequentially per chapter. E.g. "definition 3.37.4" refers to clause 4 of definition 3.37, which is contained in chapter 3.
- The symbol "□" is used to mark the end of definitions, theorems, proofs, examples, etc..
- In definitions and theorems phrases like "let x be an element of V" are abbreviated to "let $x \in V$", etc..
- This thesis contains many proofs of properties of the form $x \sqsubseteq y$, where x and y are elements of a partially ordered set $(C,\sqsubseteq)$. These proofs are given by means of a sequence $a_0,\ldots,a_n$ such that
  - $a_0 = x$
  - for all i: $0 \leq i < n$: $a_i \sqsubseteq a_{i+1}$ or $a_i = a_{i+1}$
  - $a_n = y$.

  We present these proofs in the form

  a_0
  $\sqsubseteq$ {hint why $a_0 \sqsubseteq a_1$}
  a_1
  ...
  a_{n-1}
  $\sqsubseteq$ {hint why $a_{n-1} \sqsubseteq a_n$}
  a_n .

  Proofs of implications of the form $x \Rightarrow y$ are presented in the same way. This way of presentation has been taken from [Dijkstra 3].
- Universal and existential quantification are denoted by the symbols "$\underline{A}$" and "$\underline{E}$", respectively. The symbol "|" separates domain, auxiliary condition, and quantified expression, e.g. $(\underline{A}\ x \in \mathbb{N} \mid x > 7 \mid x > 3)$. A similar notation is used for lambda expressions, e.g. the expression $(\lambda x \in V \mid x)$ denotes the identity function with domain V. In many cases domain indications are omitted when they are clear from context.

- Apart from logical expressions at the meta level we will also encounter logical expressions as elements of formal languages, e.g. in the "rule conditions" defined in chapter 2 and the condition language defined in section 3.2. Although we maintain a strict separation between these language levels we use the same set of logical symbols to form expressions. It can always be determined from context to which level an expression belongs.

Some additional notational conventions will be given in sections 2.1.2 and 2.2.2, and in notes following some definitions.

CHAPTER 1
ON FORMAL DEFINITIONS OF PROGRAMMING LANGUAGES

In this chapter we consider the role of formal definitions of programming languages, we formulate some principles and criteria regarding their use, and we motivate choice and form of the definition methods used in chapters 2 to 5.

Definitions of programming languages still have not reached the status of definitions in other branches of mathematics. Although it is generally acknowledged that definitions should be exact, complete and unambiguous, the obvious means mathematics offers to achieve these goals - viz. formalization - still has not been generally accepted. This is regrettable, as a formal definition of a programming language can be of considerable value to designers, programmers and implementers. Let us consider these categories separately:

- Formalization of a language at its design stage can help to expose and remove syntactic and semantic irregularities. If the formalism is based on solid mathematical theory it can also help to evaluate design alternatives.
- Although the formal definition of a programming language may be too complex for programmers, it can be used to develop specialized programming tools, such as proof rules or theorems concerning certain program structures (see e.g. the "Linear Search Theorem" in [Dijkstra 2]).
- A formal definition of a programming language can be used to develop exact, complete and unambiguous implementation specifications.

When we consider the present situation we must conclude that these potential possibilities have only partly been realized. A formalism like context-free grammars, which can be used to specify part of the syntax of programming languages, has gained almost universal acceptance. Although we shall not go into a detailed analysis of this success, influential factors seem to have been that context-free grammars can provide exact and unambiguous language

specifications, that they are relatively simple and amenable to mathematical treatment, that they have been used in the definition of a major programming language (ALGOL 60) before implementations of that language existed, and that they can be used to derive parts of implementations - viz. parsers - systematically and even automatically.

Formalization of context-dependent syntax and semantics has been less successful, however. On the one hand, for context-dependent syntax we find formalisms like van Wijngaarden grammars [van Wijngaarden]. These provide exact and complete syntactic specifications, are of some use in language design, but provide little or no support for implementations. On the other hand we find formalisms like attribute grammars [Knuth], which have mainly been used in compiler specifications and consequently suffer from over-specification and implementation bias when used for definition purposes. Formalization of semantics has long been a very complex affair. Gradually some usable formalisms have emerged, such as denotational semantics [Stoy] and axiomatic methods [Hoare 1, Dijkstra 2]. These methods are gaining influence on both language design [Tennent] and programming methodology [Dijkstra 2], but have little affected implementations, which are still based on informal operational interpretations of programming languages. As a general remark we can add that both formalization of context-dependent syntax and formalization of semantics have often been used only descriptively, i.e. to describe languages defined in some other way rather than to define languages. See [Ashcroft] for an illuminating discussion of this subject.

Apparently, if we want to improve the situation just sketched, we should adhere to the following principles.

- Just as in other parts of mathematics, the formal definition of a programming language should be the only source of information concerning that language. In the terminology of [Ashcroft], it should be used prescriptively rather than descriptively.
- Formal definitions should be based on well-founded and well-developed mathematical theory. The availability of such theory facilitates both language design and derivation of additional information about the defined objects.

- Overspecification should be avoided. Language definitions often contain too much irrelevant detail, which makes it difficult to isolate the essential properties.
- As a special case of the preceding principle, implementation bias should be avoided. Language constructs are often designed with a particular implementation in mind, which pervades their formal definition. As in the previous case this makes it difficult to isolate the essential properties of the constructs, but it may also block the way to completely different and unenvisaged implementations.
- Last but not least, we should keep in mind that programming languages are artefacts and that we are free to design them in such a way that they obtain a simple syntactic and semantic structure.

Let us now turn to the question what formalisms to use in our compiler correctness project. From the preceding discussion it will be clear that existing formalisms only partially conform to the principles we have formulated. The context of the project does not allow for development of new formalisms with supporting theory, which is a task of formidable size and complexity. Therefore we will content ourselves with adaptation of existing formalisms by means of simplification, providing better foundations, etc..

As far as context-dependent syntax is concerned, most of the formalisms proposed, such as van Wijngaarden grammars [van Wijngaarden], production systems [Ledgard], dynamic syntax [Ginsburg], offer little opportunity for adaptation in the sense mentioned above. The best candidate is the method of attribute grammars [Knuth], which has proven to be very useful in compiler construction, but which contains too much implementation-oriented aspects for language definition. In chapter 2 we will develop a version of attribute grammars which is primarily aimed at language definition and which is free from implementation considerations.

Selection of a suitable semantic definition method is more complicated. In the literature on program semantics there has emerged a kind of trichotomy into operational, denotational, and axiomatic methods. Roughly speaking, these methods can be characterized as follows:

- Operational methods relate the meaning of programs to state transitions of a more or less abstract machine; see e.g. [Wirth 1, Wegner].
- In denotational semantics the meanings of language constructs are explicated in terms of mathematical objects like functions. The main part of a denotational language definition consists of a set of semantic equations. The underlying theory guarantees existence of solutions of these equations; see e.g. [Stoy, de Bakker].
- Axiomatic methods are based on the fact that a set of states of a computation can be characterized by a logical formula in terms of program variables. The meaning of a language construct, especially a statement, can be defined by means of a relation between such formulae. [Floyd, Hoare 1, Dijkstra 2].

In the literature the opinion prevails that operational, denotational and axiomatic methods are most suited for implementers, language designers, and programmers, respectively. In our opinion this is a misconception, at least as far as suitability for implementers is concerned. In the computational models of operational definitions too many implementation decisions have already been made, and too much irrelevant detail has crept in. These definitions conflict with the principles of avoiding overspecification and implementation bias formulated earlier. Because of this we have decided not to base our work on operational definitions. Other considerations in the choice of a definition method have been the following:

- Axiomatic and denotational definitions are the only methods that avoid overspecification and implementation bias.
- The theory of denotational semantics is well developed. Although the method is suited for language design based on mathematical principles [Tennent], it has mainly been used descriptively. The fact that "everything" can be described denotationally does not help to obtain simple language designs.
- Axiomatic methods have not often been used as definitions. Usually they are considered as a proof system subsidiary to some other definition (operational, denotational, or informal). This somewhat secundary status conflicts with the original aims of [Hoare 1, Dijkstra 2].

- Some early experiments we have taken, see e.g. [Hemerik], suggested that implementation proofs based on axiomatic definitions would be simpler than proofs based on denotational definitions.
- The claim that axiomatic definitions provide sufficient information to derive implementations from has never been justified in practice. The literature contains hardly any references on this subject.

These considerations have led us to the decision to base our work in compiler correctness on an axiomatic method. Of those methods, predicate transformers [Dijkstra 1, Dijkstra 2] provided most grip on the subject. But even though this method has been developed sufficiently for programming purposes, its use in compiler construction required a more elaborate theoretical framework, to the extent that it has become one of the main topics of this thesis.

CHAPTER 2
FORMAL SYNTAX AND THE KERNEL LANGUAGE

2.0. Introduction

In chapter 1 we have formulated some principles regarding formal definition of programming languages. In this chapter we will apply these principles to the formal definition of the syntax of the kernel language. Our aim is to investigate how the syntax of a programming language can be specified in a manner that is devoid of implementation aspects. The discussion is based upon two well-known (though not always well-understood) formalisms, viz. context-free grammars and attribute grammars.
In section 2.1 we first recollect some definitions concerning context-free grammars and related notions, and we describe the way in which we will present context-free grammars in the remainder of this thesis. Subsequently, we point out how even in the case of such a simple and elegant formalism implementation concerns may easily creep in and influence both the definition and the definiendum. The main purpose of this section, however, is to prepare for the discussion of attribute grammars in section 2.2, which proceeds along similar lines. Traditional definitions of attribute grammars have been very implementation oriented, and the language definitions in which they have been used even more. In section 2.2 we present a definition of attribute grammars that is primarily aimed at language specification, and that is free of implementation considerations. The addition of implementation considerations relates our version to the traditional version.
Finally in section 2.3 the formalism is applied to the syntax of the kernel language, resulting in a clear and concise language specification.

At a first superficial glance it may seem that this chapter does not contain much news, since attribute grammars have been used before to define the syntax of programming languages. The novelty mainly resides in the separation of the implementation concerns from the aspects

essential to language specification, and in the simplicity resulting from it.

> "Qu on ne diſe pas que Je n ay rien dit de nouueau; la diſpoſition des matieres eſt nouuelle."
>
> Pascal, Pensées, 22.

2.1. Context-free grammars

2.1.1. Definition of context-free grammar and related notions.

Definition 2.1 {context-free grammar}

A context-free grammar G is a 4-tuple (V_N, V_T, P, Z), where

- V_N is a finite set.
- V_T is a finite set.
- $V_N \cap V_T = \emptyset$.
- P is a finite subset of $V_N \times (V_N \cup V_T)^*$.
- $Z \in V_N$.

□

V_N is the nonterminal vocabulary of G.
V_T is the terminal vocabulary of G.
$V_N \cup V_T$ is the vocabulary of G.
P is the set of production rules of G.
Z is the start symbol of G.

Definition 2.2 {>>, +>>, *>>}

Let $G = (V_N, V_T, P, Z)$ be a context-free grammar, and let $V = V_N \cup V_T$.
On V^* the relation >> is defined by:
For all $A \in V_N$, $\alpha, \beta, \gamma \in V^*$:

$$\beta A \gamma >> \beta \alpha \gamma \quad \text{fif } (A, \alpha) \in P .$$

The relation +>> is the transitive closure of >>.
The relation *>> is the reflexive and transitive closure of >>.

□

Definition 2.3 {L, language generated by a cfg}
Let $G = (V_N, V_T, P, Z)$ be a context-free grammar, and let $V = V_N \cup V_T$.

1. The function $L: V^* \to P(V_T^*)$ is defined by:

   For all $v \in V^*$: $L_G(V) = \{w \in V_T^* \mid V \ast\!\!>\!\!> w\}$.

2. The language generated by G, denoted L(G), is the set $L(Z)$.

□

Informally, a string $w \in V_T^*$ is an element of L(G) if it can be obtained by means of a systematic rewriting process on elements of V^* that begins with the start symbol Z and in which repeatedly a left-hand part of a production rule is replaced by a right-hand part until no non-terminal remains. The essentials of this rewriting process can be recorded by means of a derivation tree. The notion of a derivation tree is formalized by the following three definitions which are relative to a context-free grammar $G = (V_N, V_T, P, Z)$.

Definition 2.4 {derivation tree}
The predicate D(t,X) {t is a derivation tree with root X} is defined recursively by

$$
\begin{aligned}
D(t,X) \Leftrightarrow\ & (X \in V_T \text{ and } t = X) \\
& \text{or} \\
& (X \in V_N \text{ and } (\underline{E}\ X_1,\ldots,X_n,t_1,\ldots,t_n \mid \\
& \qquad (X,\langle X_1,\ldots,X_n\rangle) \in P \text{ and } \\
& \qquad \bigwedge_{i=1}^{n} D(t_i,X_i) \text{ and } \\
& \qquad t = (X,\langle t_1,\ldots,t_n\rangle) \\
& \qquad) \\
&) .
\end{aligned}
$$

DT is the set of all derivation trees, i.e. $DT = \{t \mid (\underline{E}\ X \mid D(t,X))\}$.
□

Definition 2.5 {frontier}
The function $f: DT \to V_T^*$ {frontier of a derivation tree} is defined recursively by

$$f(t) = \langle t \rangle \quad \text{if } t \in V_T$$

$$f((X,\langle t_1,\ldots,t_n\rangle)) = f(t_1) \oplus \ldots \oplus f(t_n)$$

where $\oplus$ is the concatenation operator.

□

Definition 2.6 {full derivation tree for a string}
The predicate FD: $DT \times V_T^* \rightarrow Bool$ is defined by

$$FD(t,w) \Leftrightarrow D(t,Z) \text{ and } f(t) = w \ .$$

□

Theorem 2.7
Let $G = (V_N,V_T,P,Z)$ be a context-free grammar, and let $V = V_N \cup V_T$.

1. For all $X \in V$, $w \in V_T^*$, $(X \ast\!\!>\!\!> w) \Leftrightarrow (\underline{E}\ t \in DT \mid D(t,X) \text{ and } f(t) = w)$.
2. For all $w \in V_T^*$, $(w \in L(G)) \Leftrightarrow (\underline{E}\ t \in DT \mid FD(t,w))$.

□

Proof
Omitted.
□

Definition 2.8 {ambiguity}
A context-free grammar $G = (V_N,V_T,P,Z)$ is ambiguous fif

$$(\underline{E}\ w \in V_T^* \mid (\underline{N}\ t \in DT \mid FD(t,w)) > 1) \ .$$

□

2.1.2. Presentation

The definitions given in section 2.1.1 are sufficient to characterize context-free grammars as formal systems. For practical purposes, however, it will be convenient to use a somewhat more redundant notation and to "prune" the less interesting parts of a large grammar. In this section we will describe the way in which we will present context-free grammars in the remainder of this thesis.

Often a considerable part of a context-free grammar is devoted to the definition of rather uninteresting constructs like identifiers,

constants, etc. The syntax of identifiers e.g. requires the following production rules

Id → Letter
Id → Id Letter
Id → Id Digit

Letter → a
⋮
Letter → z

Digit → 0
⋮
Digit → 9

merely to define identifiers as sequences of letters and digits starting with a letter. In order to shorten the grammar we can perform the following transformations.

- Remove the production rules for Id, Letter and Digit from the set of production rules.
- Remove the nonterminals Letter and Digit from the set of nonterminals.
- Introduce two subsets of V_T by

  Letter = {"a",...,"z"}
  Digit = {"0",...,"9"} .
- Extend the definition of the relation >> with:

  For all w ∈ Letter(Letter ∪ Digit)*: Id >> w .

The net effect of these transformations is a significant reduction of the number of production rules, whereas L(Id) remains the same (viz. Letter(Letter ∪ Digit)*). In the transformed grammar the nonterminal Id acts like a terminal. We will call such nonterminals pseudo terminals.

We will now describe how context-free grammars (transformed as above) will henceforth be presented.

- Nonterminals will be denoted by sequences of letters and digits starting with a capital letter. The set V_N will be given by enumeration; e.g.

  V_N = {Stat,Var,Expr,Id}

- The set V_T of terminals will be defined as the union of a finite number of sets, each of which is given by enumeration. In these enumerations the individual terminal symbols will be enclosed between quotes; e.g.

  Letter = {"a","b","c"}
  Digit = {"0","1"}
  Token = {":=","+","*","div"}
  V_T = Letter ∪ Digit ∪ Token

- The set of pseudo terminals (a subset of V_N) will be given by enumeration. The corresponding sublanguages will be given as set-theoretical expressions; e.g.

  L(Id) = Letter(Letter ∪ Digit)* .

- The set of production rules will be given by enumeration. Each element of the enumeration is presented in the format: a rule number, an element of V_N, the symbol ::=, an element of V^*, the symbol ■.
  E.g.

  1. Prog ::= |[Dec ; Stat]| ■

The first example of a context-free grammar presented in the way above is given in section 2.2.3.

2.1.3. Implementation concerns

A language specification by means of a context-free grammar $G = (V_N,V_T,P,Z)$ can be interpreted in two more or less complementary ways. The first interpretation, the classical one strongly suggested by definition 2.3, is that of a pure generative system by means of which any sentence of the language L(G) can be generated. The second interpretation, justified by theorem 2.7.2, is that of an accepting

mechanism: a given string $w \in V_T^*$ is an element of L(G) fif it is possible to construct a full derivation tree t: FD(t,w).
From a formal point of view the two interpretations are equivalent but for practical purposes important differences may result. The second interpretation is closely related to the problem of constructing a parser for L(G), a mechanism that attempts to construct a t: FD(t,w) for any $w \in V_T^*$ it receives as input. Several efficient parsing methods exist, such as LL(1), SLR(1), LALR(1), but their application usually requires the grammar to be in some special form. The danger with the second interpretation is that the language designer presents his grammar in a form that favours a certain parsing method. Such a premature choice may not only preclude the application of a different parsing method, it may also have a detrimental effect on other aspects of the formal specification and thereby on the language design itself. The following example may help to clarify this point.

Example

Let us consider the formal specification of a programming language that contains statements and in which sequential composition by means of ";" is one of the structuring mechanisms. Presumably a context-free grammar for this language contains a nonterminal S and some production rules of the form $S \rightarrow \alpha$ to define the syntactic category of statements. One of those production rules could be

(1) $S \rightarrow S;S$

which expresses that sequential composition of two statements by means of ";" results in a statement. Usually such a rule is disallowed because it leads to syntactic ambiguities. Instead a new syntactic category "statement list" is introduced by means of a nonterminal SL and a pair of production rules like

(2) $\begin{cases} SL \rightarrow S \\ SL \rightarrow SL;S \end{cases}$

or

(3) $\begin{cases} SL \rightarrow S \\ SL \rightarrow S;SL \end{cases}$

where the choice between (2) and (3) is often influenced by considerations of the kind that (2) reduces the stack size in bottom-up parsers or that (3) has no left-recursion. The desire to use an LL(1) parser may even lead to the following form:

$$(4)\quad \begin{cases} SL \rightarrow S\ RSL \\ RSL \rightarrow \varepsilon \\ RSL \rightarrow ;\ S\ RSL \end{cases}$$

The disadvantages of (2), (3) and (4) with respect to (1) are obvious: more nonterminals and production rules are required to define the same language and the simplicity and elegance of (1) are lost. The situation becomes even worse when we take other aspects of the formal specification into account, such as semantics. The semantics of a statement can be defined by means of a function f that maps a statement into its "meaning" (e.g. a predicate transformer or a state transformation). Form (1) leads to a defining clause like $f(s_1;s_2) = f(s_1) \circ f(s_2)$ in which syntax and semantics neatly match. Thanks to the associativity of function composition the syntactic ambiguity does not result in semantic ambiguity. Forms (2), (3) and (4) on the other hand either require the introduction of additional functions for syntactic categories that serve no semantic purpose, or the introduction of "abstract syntax" [McCarthy, Bjørner] which adds a level of indirection to the specification.

The objection could be raised that use of form (1) in a language specification complicates the implementation of that language since the ambiguous grammar has to be transformed into one that suits a particular parsing method. This is not always true however; e.g. a parser generator of the LR-family will generate a parser with a state containing the items [S → S;S •] and [S → S • ;S]. This state has a shift-reduce conflict for the symbol ";". The conflict can be resolved in several ways. Resolving in favour of "reduce" will result in a deterministic parser that yields left-associative derivation trees for ambiguous constructs; resolving in favour of shift will result in a parser that yields right-associative derivation trees. It is also possible to resolve the conflict nondeterministically during parsing; such a nondeterministic parser may yield any possible derivation tree

for an ambiguous construct. For none of these solutions any transformation of the grammar is required.
□

Earlier we have formulated the general principle that language specifications should not be influenced by the requirements of particular techniques. Application of this principle in the context of context-free syntax specification means that in a context-free grammar used as a language specification no commitment to a particular parsing method should be made. The grammar should be in a form that supports the definition of semantics, thus promoting simplicity and clarity. This does not mean to say that in language design implementation aspects should be ignored, however. It may be advantageous to design a language in such a way that it belongs to the class of LL(1)-languages, but the grammar used in its formal specification should first of all be oriented towards the specification of semantics and not towards the LL(1) parsing method.

2.2. Attribute grammars

2.2.0. Introduction

In section 2.1 we have seen that the generation of a string w of the language L(G) defined by a context-free grammar $G = (V_N, V_T, P, Z)$ can be considered as a rewriting process on elements of $(V_N \cup V_T)^*$. The essential property is that replacement of a nonterminal A by a string α satisfying $(A,\alpha) \in P$ may be performed regardless of the context in which A occurs. Consequently the form of a terminal production of A is completely independent of the context in which it occurs. For most nontrivial languages however properties of a construct and of its context may influence each other. Typical examples of these context-dependent properties are types and collections of definitions in force.

A popular formalism for the description of context dependencies is that of attribute grammars, introduced in [Knuth] and discussed in many places in the literature (see [Räihä] for an extensive bibliography). Usually an attribute grammar is viewed as a specification of

a computation to be performed on derivation trees. The idea is that the nodes of a derivation tree for a string can be supplied with "attributes" the values of which are determined by functions applied to attributes of surrounding nodes. The (partial) order in which these evaluations are to be performed is indicated by classifying the attributes as "inherited" or "synthesized" respectively. Most of the literature on attribute grammars is concerned with the design of efficient evaluation strategies, the automatic generation of evaluators and their use in compilers.
In the form just sketched attribute grammars have proved to be very useful as compiler specifications. They have also been used in language definitions. For the latter purpose, however, we re-encounter in a magnified form the problem of implementation bias discussed in section 2.1.3. As with context-free grammars there is the danger of orientation towards a particular parsing method for the construction of derivation trees. In addition there is the danger of orientation towards a particular evaluation strategy. The fact that by a proper classification of attributes as inherited of synthesized an efficient traversal scheme for a "tree-walking evaluator" can be obtained may be important for implementations; for language definitions the only things that matter are the relations that hold between attributes of adjacent nodes. For the latter purpose we do not need the machinery of computation on derivation trees at all; the simple notion of a parameterized production rule suffices.

There is still a second kind of overspecification involved however. The attributes are used to encode contextual information concerning types, collections of defined names, parameter correspondence, etc.. Judging from the literature the choice of a suitable formalism in which to express these properties appears to be a problem. Approaches vary from undefined operations with suggestive names [Bochmann] via more or less abstract pieces of program and data structures [Ginsburg] to formulations in terms of mathematical objects like sets, tuples, sequences, mappings, etc. [Simonet, Watt]. Even in the latter case operations are often only defined verbally due to the fact that it is difficult to express them in terms of the chosen domains and their standard operations. [Simonet] is a typical example.

The essence of the problems mentioned above is that attribute domains and operations are defined by giving an implementation of them, either in terms of mathematical objects or in terms of a programming language, but in both cases in terms of a model, and such an approach invariably introduces too many irrelevant implementation details: it is over-specific. In this respect there is a great analogy with the specification of abstract data types, or rather: the problem of the specification of an attribute system is the same as that of the specification of an abstract data type. In both cases we are not interested in any particular model or implementation of the objects and operations. All that matters are relations that hold between them and in order to determine these all we need is a way to derive them from a given set of basic properties. In other words: all we need is a proof system with a set of axioms specific to the attribute domains under consideration.

We have now isolated the aspects of an attribute grammar that are essential for language definition: a context-free grammar with parameterized production rules and a proof system to derive properties of these parameters from given axioms. In section 2.2.1 we will develop a formal system based on these aspects. Section 2.2.2 deals with the presentation of such a system in a readable form. Section 2.2.3 contains an example to illustrate various notions and the power of the formalism. Section 2.2.4 deals with implementation concerns and relates our version of attribute grammars to the traditional version.

2.2.2. Definition of attribute grammar and related notions

The first concept we introduce is that of an attribute structure, which is very similar to an algebraic specification of an abstract data type in the sense of [Goguen, Guttag]. Its most important component is a set AX of axioms. The expressions occurring in these axioms are formed from a set B of variables and a set F of function symbols; nullary function symbols serve as constants. Each expression has a certain domain ("sort" in the terminology of [Goguen] or "type name" in programming language terminology) which is determined recursively from the signature sf of function symbols and the signature sb of

variables. The set of domains D is also a component of the attribute structure. Attribute structures are defined in definition 2.9.

The attribute structures used in attribute grammars are of a special kind called boolean attribute structures. They contain the distinguished domain *Bool* corresponding to boolean expressions and they are defined relatively to a logic L, which we assume to have been predefined. Boolean attribute structures are defined in definition 2.10.

We are aware of the fact that definitions 2.9 and 2.10 still contain some gaps that might cause problems in more fundamental studies. For our purposes, which are of a more practical nature, these definitions will turn out to be sufficiently precise.

Definition 2.9 {attribute structure}
An attribute structure A is a 7-tuple (D,F,B,sf,sb,se,AX) where

- D is a set.
- F is a set.
- B is a set.
- $B \cap F = \emptyset$.
- $sf \in F \to D^* \times D$.
- $sb \in B \to D$.
- Let E be the set of expressions over elements of F and B {see note 1 below}.
- $se \in E \to D$.
- AX is a set of formulae of the form $e_1 = e_2$, where $e_1, e_2 \in E$ such that $se(e_1) = se(e_2)$.

□

D is the set of domains of A.
F is the set of function symbols of A.
B is the set of attribute variables of A.
sf is the function signature of A.
sb is the variable signature of A
se is the expression signature of A.
AX is the set of nonlogical axioms of A.

Note 1

We will not go into the details of the syntactic structure of elements of E or the definition of se. We assume that se has been defined by means of sf, sb, and recursion on the syntactic structure of expressions in the usual way.

E.g.: for all $b \in B$: $se(b) = sb(b)$.

for all $f \in F$, $e_1,\ldots,e_n \in E$:

if $sf(f) = (se \circ \langle e_1,\ldots,e_n \rangle, d)$, then $se(\langle f,e_1,\ldots,e_n \rangle) = d$.

□

Note 2

For the elements of AX universal quantification over all attribute variables occurring in them is assumed.

□

Note 3

We assume that some usual classical first order predicate logic L has been defined previously.

□

Definition 2.10 {boolean attribute structure}

An attribute structure $A = (D,F,B,sf,sb,se,AX)$ is a boolean attribute structure fif

- D contains the distinguished domain *Bool*
- F contains the function symbols of L
- for each function symbol of L: sf specifies the usual signature {i.e. $sf(true) = (\varepsilon,Bool)$, $sf(\wedge) = (\langle Bool,Bool \rangle,Bool)$, etc.}
- for each $a \in AX$: $se(a) = Bool$.

□

In the forthcoming sections we will often need the set of all expressions with a certain domain. This need motivates the following definition:

Definition 2.11 {$\underline{D}$, set of expressions with domain D}

For all $D \in D$, $\underline{D}$ denotes the set of expressions e over $F \cup B$ such that $se(e) = D$.

□

Definition 2.12 {attribute grammar}
An attribute grammar AG is a 6-tuple (V_N, V_T, Z, A, sv, R) where

- V_N is a finite set.
- V_T is a finite set.
- $V_N \cap V_T = \emptyset$.
- $Z \in V_N$.
- A is a boolean attribute structure, say $A = (D,F,B,sf,sb,se,AX)$.
- $sv \in V_N \to D^*$ such that $sv(Z) = \varepsilon$.
- Let $ANF = \{(v,\underline{x}) \in V_N \times B^* \mid sv(v) = sb \circ \underline{x}\}$.
  R is a finite set of pairs (rf,rc), where
  - $rf \in ANF \times (ANF \cup V_T)^*$
  - rc is an expression over the attribute variables in rf and over F such that $se(rc) = \mathit{Bool}$.

□

V_N is the nonterminal vocabulary of AG.
V_T is the terminal vocabulary of AG.
Z is the start symbol of AG.
sv is the nonterminal signature of AG.
ANF is the set of attributed nonterminal forms of AG.
R is the set of grammar rules of AG.
If $(rf,rc) \in R$, then

rf is the rule form of (rf,rc)
rc is the rule condition of (rf,rc).

An attribute grammar can be seen as a context-free grammar with parameterized nonterminals and production rules. Like a context-free grammar it contains a set V_N of nonterminals, a set V_T of terminals, and a start symbol $Z \in V_N$. Unlike context-free grammars, the nonterminals have some parameters - "attributes" - associated with them. For each nonterminal the number and domains of its attributes are determined by the nonterminal signature sv. Likewise, production rules are parameterized. Grammar rules, as we call them, are pairs (rf,rc) where rf is a rule form and rc is a rule condition. From a rule form production rules can be obtained by means of uniform substitution of expressions for the attribute variables. The number and domains of expressions should be in accordance with the signature of nonterminals

(definition 2.13). Nonterminals with expressions substituted for attribute variables are called attributed nonterminals (definition 2.14). The process just outlined requires a definition of substitution in rule forms etc. (definition 2.15). The essential property of attribute grammars is that the expressions to be substituted in a rule form rf must satisfy the rule condition; stated more precisely: that the rule condition with expressions substituted for attribute variables is derivable from the axioms of the attribute structure (definition 2.16).

The short summary given above is intended as clarification for definitions 2.12-2.16. The remaining definitions are very similar to those for context-free grammars.

Definition 2.13 {es, expression sequences corresponding to a domain sequence}

For all $\underline{d} \in D^*$:

$$es(\underline{d}) = \{\underline{e} \mid \underline{e} \text{ is a sequence of expressions over } F, \ dom(\underline{d}) = dom(\underline{e}), \ \underline{d} = se \circ \underline{e}\}$$

□

Definition 2.14 {AN, attributed nonterminals}

$$AN = \{(v,\underline{e}) \mid v \in V_N \text{ and } \underline{e} \in es(sv(v))\} .$$

Definition 2.15 {substitution in rule conditions, attributed nonterminal forms, terminals, rule forms}

Let $\underline{x} = \langle x_1,\ldots,x_n\rangle \in B^*$ such that the x_i are pairwise different.
Let $\underline{e} = \langle e_1,\ldots,e_n\rangle \in es(sb \circ \underline{x})$.

1. For all rule conditions rc, $rc\frac{\underline{x}}{\underline{e}}$ is defined as usual.
2. For all $(v,\langle y_1,\ldots,y_k\rangle) \in ANF$ such that $\{y_1,\ldots,y_k\} \subseteq \{x_1,\ldots,x_n\}$:

$$(v,\langle y_1,\ldots,y_k\rangle)\frac{\langle x_1,\ldots,x_n\rangle}{\langle e_1,\ldots,e_n\rangle} = (v,\langle e_{i_1},\ldots,e_{i_k}\rangle)$$

where, for j: $1 \le j \le k$: i_j is such that $x_{i_j} = y_j$.

3. For all $v \in V_T$: $v\frac{x}{e} = v$.

4. For all rule forms $(u_0,\langle u_1,\ldots,u_k\rangle)$:

$$(u_0,\langle u_1,\ldots,u_k\rangle)\frac{x}{e} = (u_0\,\frac{x}{e},\langle u_1\,\frac{x}{e},\ldots,u_k\,\frac{x}{e}\rangle) .$$

□

Definition 2.16 {pr, set of production rules derivable from a grammar rule}

For all $r = (rf,rc) \in R$:

- Let $\underline{x} \in B^*$ contain each attribute variable of rf exactly once.

- $pr(r) = rf\frac{x}{e}$ $\quad \underline{e} \in es(sb\circ\underline{x})$ and $AX \vdash_L rc\frac{x}{e}$

□

Note

In definition 2.16 we used the notation $AX \vdash_L rc\frac{x}{e}$ for provability in L of rc from AX. In the sequel we will abbreviate this to $\vdash rc\frac{x}{e}$. This should cause no confusion as other occurrence of the symbol "$\vdash$" will always be indexed.

□

Definition 2.17 {>>, +>>, *>>}

For all $A \in AN$, $\alpha,\beta,\gamma \in (AN \cup V_T)^*$:

- $\beta A\gamma >> \beta\alpha\gamma$ $\quad$ fif $(\underline{E}\ r \in R \mid (A,\alpha) \in pr(r))$.

+>> is the transitive closure of >>.

*>> is the reflexive and transitive closure of >>.

□

Definition 2.18 {L, language generated by an attribute grammar}

1. The function L: $(AN \cup V_T)^* \to P(V_T^*)$ is defined by:
   For all $v \in (AN \cup V_T)^*$: $L(v) = \{w \in V_T^* \mid v *>> w\}$.

2. The language generated by AG, denoted L(AG), is the set $L((Z,\varepsilon))$.

□

It will be clear that the power and limitations of an attribute grammar are determined by its attribute structure and its rule conditions. It

is not hard to prove that the formalism is sufficiently powerful to define any recursively enumerable language. Without further precautions it is even possible to define undecidable languages. We do not intend to impose further restrictions however. In subsequent chapters it will become clear how attribute grammars can be used to define decidable languages, not only in a theoretical but also in a practical sense.

Just as with context-free grammars the essentials of the derivation of a string $w \in L(AG)$ can be recorded by means of a tree which we will call an attributed derivation tree. The notion of an attributed derivation tree is formalized by the following definitions, which are very similar to definitions 2.4-2.6.

Definition 2.19 {attributed derivation tree}
The predicate AD(t,X) {t is an attributed derivation tree with root X} is defined recursively by:

$$
\begin{aligned}
AD(t,X) \Leftrightarrow\ & (X \in V_T \text{ and } t = X) \\
& \text{or} \\
& (X \in AN \text{ and } (\underline{E}\ X_1,\ldots,X_n,t_1,\ldots,t_n \mid \\
& \qquad (\underline{E}\ r \in R \mid (X,\langle X_1,\ldots,X_n\rangle) \in pr(r) \\
& \qquad \text{and } \bigwedge_{i=1}^{n} AD(t_i,X_i) \\
& \qquad \text{and } t = (X,\langle t_1,\ldots,t_n\rangle) \\
& \quad) \\
&)
\end{aligned}
$$

ADT is the set of all attributed derivation tree, i.e.

$$ADT = \{t \mid (\underline{E}\ X \mid AD(t,X))\}$$

□

Definition 2.20 {frontier}
The function $f: ADT \rightarrow V_T^*$ is defined recursively by:

$$f(t) = \langle t \rangle \quad \text{if } t \in V_T$$

$$f((X,\langle t_1,\ldots,t_n\rangle)) = f(t_1) \oplus \ldots \oplus f(t_n)\ .$$

□

Definition 2.21 {full attributed derivation tree for a string}
On ADT × V_T^* the predicate FAD is defined by:

$$FAD(t,w) \Leftrightarrow AD(t,Z) \text{ and } f(t) = w .$$

□

Theorem 2.22

1. For all $X \in (A_N \cup V_T)$, $w \in V_T^*$:

$$(X \ast\!\!>\!> w) \Leftrightarrow (\underline{E}\ t \in ADT \mid AD(t,X) \text{ and } f(t) = w) .$$

2. For all $w \in V_T^*$:

$$w \in L(AG) \Leftrightarrow (\underline{E}\ t \in ADT \mid FAD(t,w))$$

□

Proof
Omitted.
□

2.2.2. Presentation

As we did for context-free grammars in section 2.1.2, we will in this section describe the format in which attribute grammars will be presented henceforth.

Let AG = (V_N,V_T,Z,A,sv,R) be an attribute grammar, and
let A = (D,F,B,sf,sb,se,AX) be its attribute structure.

- D - the set of domains - will be given by enumeration. The domains will be written in italics, e.g.:

  {*Name*,*Type*,*Env*} .

- B and sb - attribute variables and their signature - will be given like variable declarations in certain programming languages. E.g. if B = $\{e_1,e_2,n\}$, $sb(e_1)$ = *Env*, $sb(e_2)$ = *Env*, sb(n) = *Name*, we write:

  e_1,e_2: *Env*; n: *Name*.

- F, sf and AX - function symbols, their signature and the non-logical axioms - will be given in the style of algebraic specifications [Goguen, Guttag]. I.e. if $f \in F$ and $sf(f) = (\langle D_1,\ldots,D_n\rangle, D)$ we write it in the form $f: D_1 * \ldots * D_n \to D$. Function symbols may be in various styles ("mixfix"): the places of the arguments are indicated by dots. E.g.:

  $[\cdot,\cdot]_D$: *Names* * *Type* → *Decs*

  $\cdot \textcircled{D} \cdot$: *Decs* * *Decs* → *Decs*

  $(\cdot,\cdot)\underline{in}_D$: *Name* * *Type* * *Decs* → *Bool*.

  For the axioms universal quantification over all free variables is assumed. Function symbols and axioms are grouped according to their "domain of interest" (cf. [Guttag]).
  In some cases it is more convenient to define the set $\underline{D}$ of all expressions e with $se(e) = D$; e.g.:

  Name = Letter(Letter ∪ Digit)* .

  We will omit the axioms for certain well-known domains such as *Int*, the domain of integer expressions.

- se - the signature of expressions - will not be mentioned explicitly.

- V_N and sv - the nonterminals and their signature - will be given by enumeration. If $X \in V_N$ and $sv(X) = \langle D_1,\ldots,D_n\rangle$ we write $X\ \langle D_1,\ldots,D_n\rangle$. E.g.:

  {Id <*Name*>,Expr <*Env*,*Prio*,*Type*>,...}

- V_T - the terminals - will be given as in section 2.1.2.

- The elements of R - the grammar rules - will be presented in the format: a rule number, an attributed nonterminal form, the symbol ::=, a sequence of attributed nonterminal forms and terminals, the symbol ■, a possibly empty sequence of formulae with domain *Bool*. The conjunction of these formulae is the rule condition of the grammar rule. E.g.:

4. Decs <d> ::= Ids <ns> : Type <t> ■
$d = [ns,t]_D$

- As in section 2.1.2 we will use pseudo-terminals in order to compress the grammar. Suppose that X <D> $\in V_N$. There will be a certain correspondence between an attribute $d \in \underline{D}$ and the set $\{w \in V_T^* \mid X \text{ <d> } *\!\!>\!\!> w\}$. That correspondence can be described by means of a relation R on $\underline{D} \times V_T^*$. Similarly to section 2.1.2 the attribute grammar can be transformed by:
  - removal of the grammar rules for X from R
  - definition of a relation R on $\underline{D} \times V_T^*$
  - extension of the relation >> by:
    for all $d \in D$, $w \in V_T^*$: X <d> >> w fif dRw

  The net effect of these transformations is that X <d> can be considered as an attributed terminal, and that $L(X \text{ <d>}) = \{w \in V_T^* \mid dRw\}$. In the presentation we will only mention the sets $L(X \text{ <d>})$ that differ from $\emptyset$.

Note

Some other notations, such as that for L {see definition 2.18} will be adapted accordingly. I.e. if X <$D_1,\ldots,D_n$> $\in V_N$ and, for i: $1 \leq i \leq n$: $d_i \in \underline{D_i}$, we write $L(X \text{ <}d_1,\ldots,d_n\text{>})$ instead of $L((X,\text{<}d_1,\ldots,d_n\text{>}))$. In addition we will write $L(X \text{ <}d_1,\ldots,D_i,\ldots,d_n\text{>})$ for $\bigcup_{d \in \underline{D_i}} L(X \text{ <}d_1,\ldots,d,\ldots,d_n\text{>})$.
□

2.2.3. Example: Satisfiable Boolean Expressions

In this section we present an example of an attribute grammar in order to illustrate some of the notions introduced in the previous sections, to illustrate the power of the formalism, and to give an impression of the parsing problem. As such it is also an introduction to section 2.2.4, which deals with implementation concerns. Not all aspects of attribute grammars are illustrated here. We pay no attention to axiomatic specifications; the first application thereof can be found in section 2.3.2. In this example we only make use of some standard

domains. Apart from *Bool* we use *Nat*, which corresponds to the language of natural numbers, and *B*, which corresponds to some language of set theory in which partial functions from natural numbers to booleans can be described by expressions like {(1,true),(2,false),(3,false)}. We consider these languages, their function symbols and axioms as given.

A well-known problem in complexity theory is the satisfiability problem [Cook 1]: Let w be a boolean expression in conjunctive normal form over the boolean variables $x_1,\ldots,x_n$, i.e. w is a conjunction of a number of factors each of which is a disjunction of the variables $x_1,\ldots,x_n$ or their negations, e.g. $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$. Find an assignment of boolean values to $x_1,\ldots,x_n$ such that n evaluates to true.

It is not hard to construct an attribute grammar that generates the language of all satisfiable boolean expressions in conjunctive normal form. As starting point we take the following context-free grammar $G = (V_N,V_T,P,Z)$:

Nonterminals

$V_N = \{Z,C,D,I\}$

Terminals

Letter = {"x"}
Digit = {"0","1","2","3","4","5","6","7","8","9"}
V_T = Letter ∪ Digit ∪ {"(",")","∧","∨","¬"}

Pseudo-terminals

{I}
$L(I)$ = Letter Digit$^+$

Start symbol

Z

Production rules

1. Z ::= C ■
2. C ::= C ∧ C ■
3. C ::= (D) ■
4. D ::= D ∨ D ■

5. D ::= I ■
6. D ::= ¬I ■

L(G) is the language of boolean expressions in conjunctive normal form. From G we will now construct an attribute grammar AG which restricts L(G) to satisfiable expressions over $x_1,\ldots,x_n$ ($n \geq 1$). With the pseudo-terminal I we associate an attribute i ∈ *Nat*, its index, such that

$L(\text{I } \langle i \rangle) = \{xv \in \text{Letter Digit}^+ \mid v \text{ is decimal representation of } i\}$.

With the nonterminals C and D we associate an attribute b ∈ *B*, which corresponds to a mapping from indices to boolean values. The correspondence between an attributed nonterminal X <b> and each of its terminal productions v is, that the set of indices of variables contained in v is dom(b), and that assignment of b(i) to x_i, for all i ∈ dom(b), satisfies v.

AG is given as follows:

Domains

{*Bool*,*Nat*,*B*}

Attribute variables

i: *Nat*;
b,b_1,b_2: *B*,

Nonterminals

V_N = {Z,C <*B*>,D <*B*>,I <*Nat*>}

Terminals

Letter = {"x"}
Digit = {"0","1","2","3","4","5","6","7","8","9"}
V_T = Letter ∪ Digit ∪ {"(",")","∧","∨","¬"}

Start symbol

Z

Pseudo-terminals

{I <*Nat*>}

For all i ∈ *Nat*:

$L(I\ \langle i \rangle) = \{xv \in \text{Letter Digit}^+ \mid v \text{ is decimal representation of } i\}$.

Grammar rules

1. Z ::= C <B> ■
   (E n: *Nat* | n > 0 | dom(b) = {1,...,n})

2. C <b> ::= C <b> ∧ C <b> ■

3. C <b> ::= (D <b>) ■

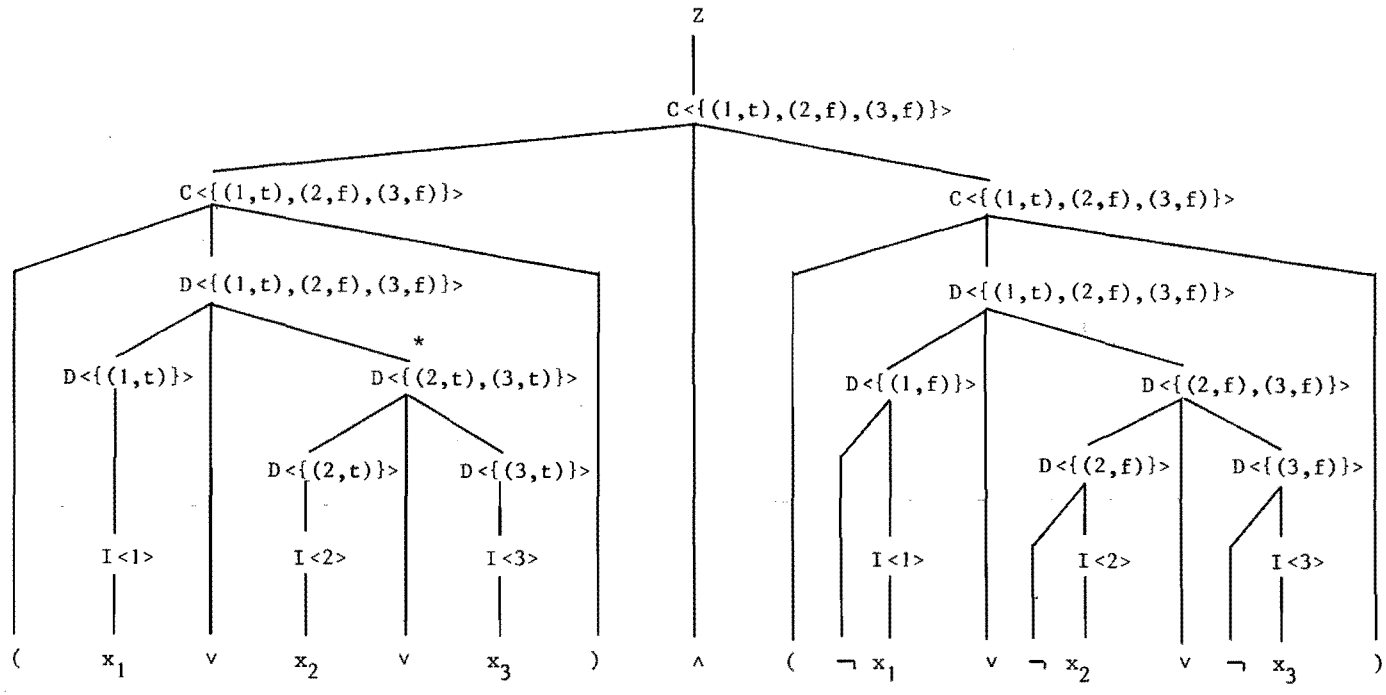
4. D <b> ::= D $\langle b_1 \rangle$ ∨ D $\langle b_2 \rangle$ ■
   $dom(b_1) \cap dom(b_2) = \emptyset$
   $dom(b_1) \cup dom(b_2) = dom(b)$
   $b/dom(b_1) = b_1$ or $b/dom(b_2) = b_2$

5. D <b> ::= I <i> ■
   b = {(i,true)}

6. D <b> ::= ¬I <i> ■
   b = {(i,false)}

The picture on page 36 corresponds to an attributed derivation tree t: $FAD(t,(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3))$. We can see from this tree that the expression $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ is satisfied by the assignment x_1,x_2,x_3 := true,false,false.

Several important observations can be made with respect to this example. The first observation is that the attribute grammar is ambiguous, i.e. there exist other attributed derivation trees for the same expression. E.g. the node marked with * might equally well be labelled with either of the attributed nonterminals D <{(2,f),(3,t)}> or D <{(2,t),(3,f)}>. This ambiguity is a consequence of the fact that the rule condition of grammar rule 4 can be satisfied in several ways that lead to identical terminal strings. In fact there exist even more attributed derivation trees for the same expression, due to the ambiguity of the context-free grammar G.



{For comment on the node marked with a "*" see page 35}

{"true" and "false" have been abbreviated to "t" and "f" respectively}

The second observation concerns the complexity of the parsing problem for this example. Since L(AG) is the set of all satisfiable boolean expressions it follows that satisfiability of a string w can be determined by an attempt to construct a t: FAD(t,w). Since the satisfiability problem is NP-complete it follows that for this example the parsing problem is NP-complete.

2.2.4 Implementation concerns

Definitions 2.12-2.21 and theorem 2.22 have been presented in a way closely resembling definitions 2.1-2.6 and theorem 2.7 in order to stress the analogies and differences with context-free grammars. Definitions 2.12-2.18 embody a generative interpretation of attribute grammars, whereas theorem 2.22.2 justifies the accepting interpretation that a string w belongs to L(AG) fif a t: FAD(t,w) can be constructed for it. Here we will concern ourselves with additional aspects that make such a construction practically feasible and that relate our view of attribute grammars to the more traditional view.

Let us first present some definitions that enable us to relate the parsing problem for attribute grammars to that for context-free grammars. All these definitions are relative to an attribute grammar $AG = (V_N, V_T, Z, A, sv, R)$.

Definition 2.23 {bs, base symbol}
$bs \in (ANF \cup AN \cup V_T) \rightarrow (V_N \cup V_T)$ such that

- for all $(v,\underline{x}) \in (ANF \cup AN)$: $bs((v,\underline{x})) = v$
- for all $v \in V_T$: $bs(v) = v$.

□

Definition 2.24 {br, base rule}
$br \in R \rightarrow V_N \times (V_N \cup V_T)^*$ such that
for all $r = ((u_0,<u_1,\ldots,u_k>),rc) \in R$:

$$br(r) = (bs(u_0),<bs(u_1),\ldots,bs(u_k)>) .$$

□

Definition 2.25 {base grammar}
The base grammar of AG is the 4-tuple (V_N,V_T,R',Z), where
$R' = \{br(r) \mid r \in R\}$.
□

Definition 2.26 {bt, base tree}
Let ADT be the set of all attributed derivation trees of AG. Let DT be the set of all derivation trees of the base grammar of AG.
$bt \in ADT \rightarrow DT$ such that

- for all $v \in V_T$: $bs(v) = v$
- for all $t_0 = (u,\langle t_1,\ldots,t_n\rangle) \in AN \times ADT^*$:

$$bt(t_0) = (bs(u),\langle bt(t_1),\ldots,bt(t_n)\rangle) .$$

□

The definitions above suggest a way to attack the parsing problem for attribute grammars. Let AG be an attribute grammar and let G be its base grammar. In order to construct a t: FAD(t,w) for a string $w \in V_T$, first construct a t': FD(t',w). Second, augment the nodes of t' with attributes in such a way that the rule conditions of the corresponding grammar rules are satisfied. If this process succeeds the result is a t: FAD(t,w) and bt(t) = t'.

The complexity of the attribution process can be reduced by imposing a partial order on attribute evaluations as follows: Each attribute position of an attributed nonterminal is classified as either inherited or synthesized. In the presentation of the attribute grammar this can be indicated by a '-' or '+' respectively. A rule form which first appeared as

$$V_0\langle \underline{x}_0\rangle ::= V_1\langle \underline{x}_1\rangle \ldots V_n\langle \underline{x}_n\rangle \blacksquare$$

then appears as

$$V_0\langle -\underline{i}_0,+\underline{s}_0\rangle ::= V_1\langle -\underline{i}_1,+\underline{s}_1\rangle \ldots V_n\langle -\underline{i}_n,+\underline{s}_n\rangle \blacksquare$$

where for each j: $0 \leq j \leq n$ the couple $\underline{i}_j,\underline{s}_j$ is a "partition" of $\underline{x}_j$. The corresponding rule condition $P(\underline{x}_0,\ldots,\underline{x}_n)$ can be transformed to an evaluation rule by writing it as:

$$\underline{s}_0,\underline{i}_1,\ldots,\underline{i}_n: P(\underline{i}_0,\underline{s}_0,\ldots,\underline{i}_n,\underline{s}_n)$$

which is to be interpreted as: "determine $\underline{s}_0,\underline{i}_1,\ldots,\underline{i}_n$ from $\underline{i}_0,\underline{s}_1,\ldots,\underline{s}_n$ such that $P(\underline{i}_0,\underline{s}_0,\ldots,\underline{i}_n,\underline{s}_n)$ holds. We see that the synthesized attributes from the left-hand part and the inherited attributes from the right-hand part must be computed from the other attributes. In order to avoid conflicts some well-formedness conditions have to be imposed. An occurrence of an attribute variable among $\underline{i}_0,\underline{s}_1,\ldots,\underline{s}_n$ is called a defining occurrence; among $\underline{s}_0,\underline{i}_1,\ldots,\underline{i}_n$ it is called an applied occurrence. In each grammar rule each attribute may have at most one defining occurrence. Furthermore, to ensure that the implied evaluation order is a partial order indeed there may be no cycles. Algorithms to verify the absence of cycles have been described in [Knuth, Jazayeri].

Thus extended our notion of attribute grammar comes quite close to the traditional notion. There is an important difference in the form of the evaluation rules, however. In our version evaluation rules are of the form

(1) $\underline{y}$: $P(\underline{x},\underline{y})$, where $P(\underline{x},\underline{y})$ is a condition,

whereas the traditional form is

(2) $\underline{y} = F(\underline{x})$, where F is a function.

Form (2) suffices for traditional applications as intended by Knuth [Knuth] where the sole purpose is to compute a function of the strings of a context-free language. The application of attribute grammars as language acceptors however hinges on the fact that rule conditions may or may not have a solution. That fact is easily catered for by form (1), whereas form (2) would require provisions to deal with partial functions, such as domain restrictions or error values, which soon proliferate through the entire grammar. Many published attribute grammars show deficiencies in this respect.

Last but not least there is the important aspect of correct implementations. We recall from chapter 0 that we have set as our goal the derivation of a correct compiler from formal definitions of both source and target language. A major subtask is the construction of a correct acceptor for the language defined by an attribute grammar. We will not concern ourselves with evaluation strategies; enough is known about that problem. What remains is the construction of the code for the

individual attribute evaluations, a significant part of the total compiler code. Our version of attribute grammars supports this task in two respects:

- the rule conditions of the grammar rules may be used directly as post-conditions for the code to be constructed;
- since attribute structures correspond to algebraic specifications of abstract data types all of the programming methodology available in that field can be applied directly to the implementation of attribute domains and their associated operations.

Here we will not elaborate on these aspects. They will be treated extensively in a subsequent report [Hemerik].

Above we have described how by addition of "implementation aspects" from our version of attribute grammar an attribute grammar in the traditional sense may be obtained. These aspects have often unnecessarily influenced and complicated language specifications. We hope to have made clear that they can, and should, be separated from language definition aspects.

2.3. Formal syntax of the kernel language

In this section we will develop the formal syntax of the kernel language. In section 2.3.1 we present as first approximation a context-free grammar. In section 2.3.2 this grammar is extended to an attribute grammar that captures all context dependent properties as well.

2.3.1. A context-free grammar for the kernel languages

The kernel language is much like the language fragment contained in [Dijkstra 2]. Roughly speaking it consists of the following ingredients:

- the statements abort, skip, multiple assignment, alternative statement, repetitive statement, block;
- integer and boolean expressions;
- explicit declaration of variables.

With the exception of blocks and declarations the constructs have the same appearance as in [Dijkstra 2]. Variable declarations are similar to those in Pascal. The rest of the grammar should speak for itself.

Nonterminals

V_N = {Prog,Block,Decs,Stat,Type,Ids,Id,Vars,Var,Exprs,Expr,Gcs, Dop,Mp,Con}

Terminals

Letter = {"a",...,"z"}
Digit = {"0",...,"9"}
Op1 = {"+","-","¬"}
Op2 = {"*","+","-","=","≠","<","≤",">","≥","∧","∨","⇒","⇔"}

Typesym = {"int","bool"}
Consym = {"true","false"}
Statsym = {"skip","abort"}
Sym = {"|[","]|","|",",",":",";", "[]","→",":=","(",")", "if","fi","do","od","var"}

V_T = Letter ∪ Digit ∪ Op1 ∪ Op2 ∪ Typesym ∪ Consym ∪ Statsym ∪ Sym.

Start symbol

Prog

Pseudo terminals

{Id,Dop,Mop,Con,Type}
L(Id) = Letter(Letter ∪ Digit)* \ (Typesym ∪ Consym ∪ Statsym)
L(Dop) = Op2
L(Mop) = Op1
L(Con) = Digit$^+$ ∪ Consym
L(Type) = Typesym

Production rules

1. Prog ::= Block ■

2. Block ::= |[var Decs | Stat]| ■

3. Decs ::= Decs , Decs ■
4. Decs ::= Ids : Type ■

5. Ids ::= Ids , Ids ■
6. Ids ::= Id ■

7. Stat ::= abort ■
8. Stat ::= skip ■
9. Stat ::= Vars := Exprs ■
10. Stat ::= Stat ; Stat ■
11. Stat ::= if Gcs fi ■
12. Stat ::= do Gcs od ■
13. Stat ::= Block ■

14. Vars ::= Vars , Vars ■
15. Vars ::= Var ■

16 Exprs ::= Exprs , Exprs ■
17. Exprs ::= Expr ■

18. Expr ::= Expr Dop Expr ■
19. Expr ::= Mop Expr ■
20. Expr ::= (Expr) ■
21. Expr ::= Var ■
22. Expr ::= Con ■

23. Var ::= Id ■

24. Gcs ::= Gcs [] Gcs ■
25. Gcs ::= Expr → Stat ■

2.3.2. An attribute grammar for the kernel language

Upon the language defined in section 2.3.1 a number of context conditions are imposed in order to exclude programs like the following:

```
|[ var x : int |
   |[ var x : bool, i : int, i,b : bool |
            x,z := 3,4;
            do true > (3 ∧ 4 * b) → b, b := 3 od
   ]|;
   b := x > 3
]|
```

Informally stated the context conditions are as follows:

- Within a declaration part of a block each variable may occur at most once.
- Each variable occurring in a statement must be declared in some surrounding block.
- Redeclaration of variables in nested blocks is allowed. This point will be reconsidered in chapter 4.
- Expressions should be well-formed with respect to priorities of operators and types of operands.
- Left part and right part of assignments should be of corresponding lengths and types.
- Within the left part of an assignment each variable may occur at most once.

For the formal rendering of the above we will introduce a number of domains and operations. Below we provide some informal explanation concerning their purpose. This explanation may help in reading the language specification but is not part of it.

Bool, *Int*:
Need no further explanation.

Prio:
Used to indicate the priorities of operators and expressions. The elements of *Prio* are those of *Int* corresponding to the numbers 1,...,7.

Type:
Used to indicate the type of expressions and variables. We take *Type* = Typesym.

Name:
Used to distinguish between various identifiers. We take *Name* = Letter(Letter ∪ Digit)*.

Names:
Used to indicate the collection of names occurring in a declaration part or in the left part of an assignment. An expression ns ∈ *Names* is either of the form $[n]_N$, where n ∈ *Name*, or of the form $ns_1 \cup_N ns_2$, where $\{ns_1, ns_2\} \subseteq$ *Names*. ns may be thought of as a bag of names, in which case the other operations $\underline{in}_N$ and $\#_N$ correspond to membership and number of occurrences respectively. From the axioms it is easy to prove that $(n \ \underline{in}_N \ ns) = (\#_N(n,ns) \geq 1)$.

Types:
Used to indicate the sequence of types corresponding to the left part or right part of an assignment. An expression ts ∈ *Types* is either a singleton of the form $[t]_T$, where t ∈ *Type*, or of the form $ts_1 \oplus_T ts_2$, where $\{ts_1, ts_2\} \subseteq$ *Types*. To compensate for the ambiguities in de production rules for Vars en Exprs there is an axiom which states associativity of $\oplus_T$.

Decs:
Used to record the essential information of a declaration part, viz. a collection of (name,type) associations. An expression d ∈ *Decs* is either of the form $[ns,t]_D$, where ns ∈ *Names* and t ∈ *Type*, or of the form $d_1 \cup_D d_2$, where $\{d_1, d_2\} \subseteq$ *Decs*. d may be thought of as a bag of pairs (name,type), in which case the operation $\underline{in}_D$ corresponds to membership and the operation $\#_D$ to number of occurrences of a name.

Env:
Used to record the environment of a construct, i.e. the essential information of all declarations occurring in blocks surrounding the construct, taking into account the nesting of blocks. An expression e ∈ *Env* is either Empty, which corresponds to an empty collection of

declarations, or of the form Ext(e,d), where e $\in$ *Env* and d $\in$ *Decs*, which corresponds to an ordered extension of an environment with a collection of declarations.

After this informal presentation reading of the grammar should pose no serious problems. The presentation follows.

Domains

{*Bool*,*Int*,*Name*,*Type*,*Prio*,*Names*,*Types*,*Decs*,*Env*}

Attribute variables

n, n_1, n_2: *Name*;
t, t_0, t_1, t_2: *Type*;
p, p_0, p_1, p_2: *Prio*;
ns, ns_0, ns_1, ns_2: *Names*;
ts, ts_0, ts_1, ts_2: *Types*;
d, d_0, d_1, d_2: *Decs*;
e, e_0, e_1: *Env*.

Operations on *Prio*

Prio = {e $\in$ *Int* | the integer value corresponding to e is an element of {1,...,7} } .

Operations on *Type*

Type = Typesym.

Operations on *Name*

Name = Letter(Letter $\cup$ Digit)*.

Operations on *Names*

$[\cdot]_N$: *Name* $\rightarrow$ *Names*
$\cdot \underbrace{N} \cdot$: *Names* $*$ *Names* $\rightarrow$ *Names*
$\cdot \underline{\text{in}}_N \cdot$: *Name* $*$ *Names* $\rightarrow$ *Bool*
$\#_N(\cdot,\cdot)$: *Name* $*$ *Names* $\rightarrow$ *Int*

$n_1 \underline{\text{in}}_N [n_2]_N = (n_1 = n_2)$

$n_1 \underline{\text{in}}_N (ns_1 \textcircled{N} ns_2) = (n_1 \underline{\text{in}}_N ns_1) \vee (n_1 \underline{\text{in}}_N ns_2)$

$\#_N(n_1,[n_2]_N) = \underline{\text{if}}\ n_1 = n_2 \rightarrow 1 \ [] \ n_1 \neq n_2 \rightarrow 0\ \underline{\text{fi}}$

$\#_N(n_1,ns_1 \textcircled{N} ns_2) = \#_N(n,ns_1) + \#_N(n,ns_2)$

Operations on *Types*

$[\cdot]_T$: *Type* → *Types*

$\cdot \oplus_T \cdot$: *Types* * *Types* → *Types*

$(ts_1 \oplus_T ts_2) \oplus_T ts_3 = ts_1 \oplus_T (ts_2 \oplus_T ts_3)$

Operations on *Decs*

$[\cdot,\cdot]_D$: *Names* * *Type* → *Decs*

$\cdot \textcircled{D} \cdot$: *Decs* * *Decs* → *Decs*

$(\cdot,\cdot) \underline{\text{in}}_D \cdot$: *Name* * *Type* * *Decs* → *Bool*

$\#_D(\cdot,\cdot)$: *Name* * *Decs* → *Int*

$(n,t_1) \underline{\text{in}}_D [ns,t_2]_D = n \underline{\text{in}}_N ns \wedge (t_1 = t_2)$

$(n,t) \underline{\text{in}}_D (d_1 \textcircled{D} d_2) = ((n,t) \underline{\text{in}}_D d_1) \vee ((n,t) \underline{\text{in}}_D d_2)$

$\#_D(n,[ns,t]_D) = \#_N(n,ns)$

$\#_D(n,d_1 \textcircled{D} d_2) = \#_D(n,d_1) + \#_D(n,d_2)$

Operations on *Env*

Empty : *Env*

Ext(·,·) : *Env* * *Decs* → *Env*

$(\cdot,\cdot) \underline{\text{in}}_E \cdot$: *Name* * *Type* * *Env* → *Bool*

$(n,t) \underline{\text{in}}_E$ Empty = false

$(n,t) \underline{\text{in}}_E$ Ext(e,d) = $(n,t) \underline{\text{in}}_D d \vee (\#_D(n,d) = 0 \wedge (n,t) \underline{\text{in}}_E e)$

Nonterminals

V_N = {Prog,Block <*Env*>,Decs <*Decs*>,Stat <*Env*>,Type <*Type*>,Ids <*Names*>,
Id <*Name*>,Vars <*Env*,*Names*,*Types*>,Var <*Env*,*Name*,*Type*>,
Exprs <*Env*,*Types*>,Expr <*Env*,*Prio*,*Type*>,Gcs <*Env*>,
Dop <*Prio*,*Type*,*Type*,*Type*>,Mop <*Type*,*Type*>,Con <*Type*>}.

Terminals

Letter = {"a",...,"z"}
Digit = {"0",...,"9"}
Op1 = {"+","-","¬"}
Op2 = {"*","+","-","=","≠","<","≤",">","≥","∧","∨","⇒","⇔"}

Typesym = {"int","bool"}
Consym = {"true","false"}
Statsym = {"skip","abort"}
Sym = {"|[","]|","|",",",":",";", "□","→",":=","(",")",
"if","fi","do","od","var"}

V_T = Letter ∪ Digit ∪ Op1 ∪ Op2 ∪ Typesym ∪ Consym ∪ Statsym ∪ Sym.

Start symbol

Prog.

Pseudo terminals

{Id <*Name*>,Dop <*Prio*,*Type*,*Type*,*Type*>,Mop <*Type*,*Type*>,Con <*Type*>,
Type <*Type*>}.

For all n ∈ *Name*:

L(Id <n>) = {n} \ (Typesym ∪ Consym ∪ Statsym) .

L(Dop <1,bool,bool,bool>) = {"⇒","⇔"}
L(Dop <2,bool,bool,bool>) = {"∨"}
L(Dop <3,bool,bool,bool>) = {"∧"}
L(Dop <4,bool,int,int>) = {"=","≠","<","≤",">","≥"}
L(Dop <5,int,int,int>) = {"+","-"}
L(Dop <6,int,int,int>) = {"*"}

L(Mop <int,int>) = {"+","−"}
L(Mop <bool,bool>) = {"¬"}

L(Con <int>) = Digit$^+$
L(Con <bool>) = Consym

L(Type <int>) = {"int"}
L(Type <bool>) = {"bool"}

Grammar rules

1. Prog ::= Block <e> ■
   e = Empty

2. Block <e_0> ::= |[var Decs <d> | Stat <e_1>]| ■
   (A n: *Name* | $\#_D(n,d) \leq 1$)
   e_1 = Ext(e_0,d)

3. Decs <d_0> ::= Decs <d_1> , Decs <d_2> ■
   $d_0 = d_1 \overset{D}{\frown} d_2$

4. Decs <d> ::= Ids <ns> : Type <t> ■
   $d = [ns,t]_D$

5. Ids <ns_0> ::= Ids <ns_1> , Ids <ns_2> ■
   $ns_0 = ns_1 \overset{N}{\frown} ns_2$

6. Ids <ns> ::= Id <n> ■
   $ns = [n]_N$

7. Stat <e> ::= abort ■

8. Stat <e> ::= skip ■

9. Stat <e> ::= Vars <e,ns,ts> := Exprs <e,ts> ■
   (A n: *Name* | $\#_N(n,ns) \leq 1$)

10. Stat <e> ::= Stat <e> ; Stat <e> ■

11. Stat <e> ::= if Gcs <e> fi ■

12. Stat <e> ::= do Gcs <e> od ■

13. Stat <e> ::= Block <e> ■

14. Vars <e,ns_0,ts_0> ::= Vars <e,ns_1,ts_1> , Vars <e,ns_2,ts_2> ■
$ns_0 = ns_1 \overset{N}{\frown} ns_2$
$ts_0 = ts_1 \oplus_T ts_2$

15. Vars <e,ns,ts> ::= Var <e,n,t> ■
$ns = [n]_N$
$ts = [t]_T$

16. Exprs <e,ts_0> ::= Exprs <e,ts_1> , Exprs <e,ts_2> ■
$ts_0 = ts_1 \oplus_T ts_2$

17. Exprs <e,ts> ::= Expr <e,p,t> ■
$ts = [t]_T$

18. Expr <e,p_0,t_0> ::= Expr <e,p_1,t_1> Dop <p_0,t_0,t_1,t_2> Expr <e,p_2,t_2> ■
$p_0 \leq p_1$
$p_0 < p_2$

19. Expr <e,p_0,t_0> ::= Mop <t_0,t_1> Expr <e,p_1,t_1> ■
$p_0 = 7$
$p_1 = 7$

20. Expr <e,p_0,t> ::= (Expr <e,p_1,t>) ■
$p_0 = 7$

21. Expr <e,p,t> ::= Var <e,n,t> ■
$p = 7$

22. Expr <e,p,t> ::= Con <t> ■
$p = 7$

23. Var <e,n,t> ::= Id <n> ■
$(n,t) \underline{\text{in}}_{\epsilon} e$

24. Gcs <e> ::= Gcs <e> [] Gcs <e> ■

25. Gcs <e> ::= Expr <e,p,t> → Stat <e> ■
$t = bool$

CHAPTER 3
PREDICATE TRANSFORMER SEMANTICS FOR THE KERNEL LANGUAGE

3.0. Introduction

In [Dijkstra 1, Dijkstra 2] it has been proposed to define the semantics of programming languages by means of so-called predicate transformers. The idea is that a set of states of a computation can be characterized by a predicate in terms of the program variables and that all relevant aspects of a statement are captured by its predicate transformer, a function from predicates to predicates. Two kinds of predicate transformers are discussed, viz. the "weakest pre-condition" wp and, to a lesser extent, the "weakest liberal pre-condition" wlp. For a certain mechanism S and a post-condition R the corresponding weakest pre-condition wp(S,R) is defined as follows (we quote from [Dijkstra 2]):

> "The condition that characterizes the set of all initial states such that activation will certainly result in a properly terminating happening leaving the system in a final state satisfying a given post-condition is called 'the weakest pre-condition corresponding to that post-condition'.".

Experience has shown that predicate transformers are a suitable vehicle for discussing issues of program correctness. When dealing with questions of semantics or implementation correctness further elaboration is required, however. Let us mention a few problems:

- As remarked by [Plotkin] the definition quoted above "is admirably clear and perfectly precise once we know what conditions and mechanisms are". In [Dijkstra 2] several mechanisms (statements) are defined but no definition of conditions is given. The result is that of the central concept of the language definition, the predicate transformer, two important aspects, its domain and its range, are left undefined. That such an omission may lead to complications can be illustrated as follows.

Let C be a set of conditions that serves as domain and range of predicate transformers. Two statements S_1 and S_2 may be considered equivalent fif for all $Q \in C$: $wp(S_1,Q) = wp(S_2,Q)$.

- In the extreme case that $C = \emptyset$ it follows that all statements are equivalent.
- In [Dijkstra 2] the predicate transformers of the statements skip and x := E are defined by wp(skip,R) = R and $wp(x := E,R) = R_{E \to x}$, respectively. In case that C contains no conditions that depend on x these statements are equivalent, otherwise they are not.

Simple and artificial as these examples may seem, they suffice to show that the degree to which statements can be distinguished depends on the "richness" of the set of conditions C, which therefore is an essential component of a language definition.

- Another point of concern is the nature of conditions. In [Dijkstra 2] the distinction between formal expressions and the objects they denote is simply done away with as a "mannerism". We just cannot afford such an attitude in language translation: the very existence of the notion of translation is based on the fact that different formal expressions may denote the same object. In the literature on predicate transformer semantics we find both an intensional point of view, where conditions are considered as elements of a formal language [Back 2, de Bakker, Milne], and an extensional point of view, where conditions are identified with the sets of states they characterize [Plotkin, Wand]. Both approaches are feasible, but each has its specific problems and there are marked differences, e.g. with regard to the lattice-theoretical framework required to deal with recursively defined conditions.
- The last remark above hints at a different problem. The definition of wp(DO,R) in [Dijkstra 2] essentially employs a fixed point approximation, although this is not stated explicitly. Since in further development of predicate transformer semantics we will not only encounter recursively defined predicates but also various kinds of recursively defined predicate transformers it seems wise

to employ a general theory to deal with recursive definitions, such as Scott's lattice theory [Scott 2].

- A final point concerns the expressibility of conditions. In general, the pre-condition of a statement will depend on the variables and expressions occurring in that statement and the language of conditions should be rich enough to express those dependencies. A good candidate for a condition language seems to be the set of first order predicates in terms of the variables and operations of a program. Yet it appears that this language is not sufficiently powerful to express pre-conditions of repetitions. [Back 1] gives a simple counter-example. Either a larger set of operations should be employed (which raises the problem of determining whether the condition language is closed under that set), or a more powerful logic should be employed such as the infinitary logic $L_{\omega_1\omega}$ [Back 1, Karp, Scott 1]. [Back 1] shows that this logic is sufficiently powerful to express the weakest pre-conditions for the language of [Dijkstra 2].

It is the purpose of this chapter to provide a firm foundation for predicate transformer semantics by presenting additional definitions and by making the connections with other branches of mathematics such as lattice theory and logic more explicit. In doing so we will also pave the way for the development of a predicate transformer semantics for language constructs other than statements in chapters 4 and 5.

The first important decision we take in this respect is to adopt an intensional view of conditions, i.e. to consider them as elements of a formal language. The grammatical tools developed in chapter 2 enable us to define very precisely the condition language to be used with a set of statements, in particular as regards the contextual properties. These properties play an important role in chapter 4, where constructs involving changes of context are considered, such as blocks and procedures. The close connections between programming language and condition language also make it easier to study Hoare-like correctness formulae and proof rules, as statements and the conditions associated with them can be considered at the same language level, and transitions between syntactic and semantic domains can be kept to a minimum.

Imposing a suitable lattice structure on the condition language requires some provisions. On the one hand we will introduce infinitary formulae and proof rules as in $L_{\omega_1\omega}$. On the other hand we will develop some theory for a special kind of lattices which we call countably-complete lattices and which provide the desired structure to deal with recursive functions of conditions. This chapter therefore has the following structure. In section 3.1 we collect both old and new results from lattice theory that will frequently be used in the current and following chapters. In section 3.2 we first define a condition language and subsequently study the predicate transformers wp and wlp in a lattice-theoretical framework. In section 3.3 we use these definitions to develop logics in the style of [Hoare 2] for proving partial and total correctness of programs.

Note

Thus far we have used the term "predicate transformer" introduced by Dijkstra. Because the notion of condition differs considerably from the notion of predicate as used in logic, we will henceforth use the term "condition transformer".

□

3.1. Some lattice theory

In this section we collect some results concerning complete partially ordered sets, (countably) complete lattices, continuous functions, fixed points, etc. These results form the basis for the study of condition transformer semantics in subsequent sections. Part of the material presented here is well-known and has mainly been included for completeness's sake. Lemmas and theorems that appear without proof have been taken over literally or with slight adaptations from [de Bakker], as have some definitions. Less-known and new results appear with full proofs. The proofs of some subsidiary results have been delegated to Appendix A.

3.1.1. General definitions

The central notions of our summary are those of a complete partially ordered set (cpo) and of a countably-complete lattice (ccl). Since both of them are special forms of partially ordered sets we begin our sequence of definitions with that of the latter notion.

Definition 3.1 {partially ordered set}
A partially ordered set is a pair $(C,\sqsubseteq)$, where C is an arbitrary set and $\sqsubseteq$ is a binary relation on C satisfying

- $(\underline{A}\ x \in C \mid x \sqsubseteq x)$ {reflexivity}
- $(\underline{A}\ x,y \in C \mid (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y)$ {antisymmetry}
- $(\underline{A}\ x,y,z \in C \mid (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z)$ {transivity}

Note
Occasionally we will also use the relations $\sqsubset$ and $\sqsupseteq$, given by $(\underline{A}\ x,y \in C \mid x \sqsubset y \Leftrightarrow x \sqsubseteq y \wedge x \neq y)$ and $(\underline{A}\ x,y \in C \mid x \sqsupseteq y \Leftrightarrow y \sqsubseteq x)$, respectively.
□

A subset of a partially ordered set may have a greatest lower bound or a least upper bound:

Definition 3.2 {glb, lub}
Let $(C,\sqsubseteq)$ be a partially ordered set, and let $X \subseteq C$.

1. $y \in C$ is called the greatest lower bound (glb) of X if
   - $(\underline{A}\ x \in X \mid y \sqsubseteq x)$,
   - $(\underline{A}\ z \in C \mid (\underline{A}\ x \in X \mid z \sqsubseteq x) \Rightarrow z \sqsubseteq y)$.

   The glb of a set X will be denoted by $\sqcap X$.
2. $y \in C$ is called the least upper bound (lub) of X if
   - $(\underline{A}\ x \in X \mid x \sqsubseteq y)$,
   - $(\underline{A}\ z \in C \mid (\underline{A}\ x \in X \mid x \sqsubseteq z) \Rightarrow y \sqsubseteq z)$.

   The lub of a set X will be denoted by $\sqcup X$.
3. The glb and lub of a sequence $\langle x_0, x_1, \ldots \rangle$ are denoted by $\prod_{i=0}^{\infty} x_i$ and $\bigsqcup_{i=0}^{\infty} x_i$, respectively.

□

Note
From the definitions above it follows that if for $X \subseteq C$ $\sqcap X$ or $\sqcup X$ exist, they must be unique.
□

Definition 3.3 {chain}
Let $(C,\sqsubseteq)$ be a partially ordered set.

1. An ascending chain in $(C,\sqsubseteq)$ is a sequence $\langle x_0, x_1, \ldots \rangle$ such that $(\underline{A}\ i \mid i \geq 0 \mid x_i \sqsubseteq x_{i+1})$.
2. A descending chain in $(C,\sqsubseteq)$ is a sequence $\langle x_0, x_1, \ldots \rangle$ such that $(\underline{A}\ i \mid i \geq 0 \mid x_i \sqsupseteq x_{i+1})$.

□

We will primarily be interested in a special kind of partially ordered sets, viz. countably-complete lattices. It will be useful to relate that notion to some better-known notions, however. Hence the following definitions.

Definition 3.4 {cpo}

A complete partially ordered set (cpo) is a partially ordered set $(C,\sqsubseteq)$ such that

- the glb of C exists,
- each ascending chain $\langle x_0,x_1,\ldots\rangle$ has a lub $\bigsqcup_{i=0}^{\infty} x_i$.

□

Definition 3.5 {uccl, dccl}

1. An upward countably-complete lattice (uccl) is a partially ordered set $(C,\sqsubseteq)$ with the property that each countable subset has a lub and each finite subset has a glb.
2. A downward countably-complete lattice (dccl) is a partially ordered set $(C,\sqsubseteq)$ with the property that each countable subset has a glb and each finite subset has a lub.

□

Definition 3.6 {ccl}

A countably-complete lattice (ccl) is a partially ordered set $(C,\sqsubseteq)$ with the property that each countable subset has both a lub and a glb.

□

Definition 3.7 {cl}

A complete lattice (cl) is a partially ordered set $(C,\sqsubseteq)$ with the property that each subset has both a lub and a glb.

□

Note

In definition 3.7 it would suffice to require that each subset $X \subseteq C$ has a lub. It can easily be shown that in that case it also has a glb, viz. $\sqcup\{y \in C \mid (\underline{A}\ x \in X \mid y \sqsubseteq x)\}$.

□

Lemma 3.8

Let $(C,\sqsubseteq)$ be a uccl.

1. C has a glb, viz. $\sqcup\emptyset$.
2. C has a lub, viz. $\sqcap\emptyset$.

□

Proof

1. $\emptyset$ is a countable subset of C, hence by definition 3.2.2:

   $(\underline{A}\ z \in C \mid (\underline{A}\ x \in \emptyset \mid x \sqsubseteq z) \Rightarrow \sqcup\emptyset \sqsubseteq z)$, i.e.

   $(\underline{A}\ z \in C \mid \sqcup\emptyset \sqsubseteq z)$ {a}

   Let $w \in C$ be such that $(\underline{A}\ z \in C \mid w \sqsubseteq z)$. Then in particular:

   $w \sqsubseteq \sqcup\emptyset$ {b}

   From a, b and definition 3.2.1 it follows that $\sqcap C = \sqcup\emptyset$.

2. $\emptyset$ is a finite subset of C. The remainder of the proof is a dual version of 1.

□

From definitions 3.4-3.7 and lemma 3.8 it follows immediately that

- every cl is a ccl;
- every ccl is both a uccl and a dccl;
- every uccl is a cpo;
- the dual of a cl is a cl;
- the dual of a ccl is a ccl;
- the dual of a uccl is a dccl.

In the sequel we will frequently make use of these relations; e.g. if we have proven a property of cpo's we will use the fact that it also holds for uccl's and that the dual property holds for dccl's.

Note

If $(C,\sqsubseteq)$ is a cpo (uccl, dccl, ccl, cl) its glb $\sqcap C$ will be written as $\perp$ {pronounced as "bottom"}. If $(C,\sqsubseteq)$ is a uccl (dccl, ccl, cl) its lub $\sqcup C$ will be written as $\top$ {pronounced as "top"}.

□

Our interest will focus on functions on lattices of the kinds just introduced. An important property of certain classes of these functions is that under a suitably defined order they also have a lattice structure. That order is defined by

Definition 3.9 {standard order on functions}
Let C_1 be a set and $(C_2,\sqsubseteq_2)$ be a partially ordered set. The standard order $\sqsubseteq$ on $C_1 \to C_2$ is defined by:

$$(\underline{A}\ f,g \in C_1 \to C_2 \mid f \sqsubseteq g \Leftrightarrow (\underline{A}\ x \in C_1 \mid f(x) \sqsubseteq_2 g(x)) .$$

□

It can easily be verified that the standard order is a partial order indeed.

Theorem 3.10
Let C_1 be a set and $(C_2,\sqsubseteq)$ be a partially ordered set. Let $\sqsubseteq$ be the standard order on $C_1 \to C_2$.

1. If $(C_2,\sqsubseteq_2)$ is a cpo, then
   - $(C_1 \to C_2,\sqsubseteq)$ is a cpo.
   - $\perp_{C_1 \to C_2} = (\lambda x \mid \perp_{C_2})$.
   - for each ascending chain $\langle f_i \rangle_{i=0}^{\infty}$ in $C_1 \to C_2$:
     $$\bigsqcup_{i=0}^{\infty} f_i = (\lambda x \mid \bigsqcup_{i=0}^{\infty} f_i(x)) .$$
2. If $(C_2,\sqsubseteq_2)$ is a uccl, then
   - $(C_1 \to C_2,\sqsubseteq)$ is a uccl.
   - for each countable $F \subseteq C_1 \to C_2$: $\sqcup F = (\lambda x \mid \bigsqcup_{f \in F} f(x))$.
   - for each finite $F \subseteq C_1 \to C_2$: $\sqcap F = (\lambda x \mid \bigsqcap_{f \in F} f(x))$.

□

Corollary 3.11
If C_1 is a set and $(C_2,\sqsubseteq_2)$ is a uccl:

- $\perp_{C_1 \to C_2} = (\lambda x \mid \perp_{C_2})$.
- $\top_{C_1 \to C_2} = (\lambda x \mid \top_{C_2})$.

□

Proof.
1. See [de Bakker].

2. 1. Let F be a countable subset of $C_1 \to C_2$. From the fact that $(C_2, \sqsubseteq_2)$ is a uccl it follows that for all $x \in C_1$:

$\bigsqcup_{f \in F} f(x) \in C_2$.

Let $g = (\lambda x \mid \bigsqcup_{f \in F} f(x))$. We show that $g = \sqcup F$.

1. $(C_2, \sqsubseteq_2)$ is a uccl

$\Rightarrow (\underline{A}\ f \in F \mid (\underline{A}\ x \in C_1 \mid f(x) \sqsubseteq_2 \bigsqcup_{f \in F} f(x)))$

$\Rightarrow$ {definition 3.9, definition of g}

$(\underline{A}\ f \in F \mid f \sqsubseteq g)$

2. Let $h \in C_1 \to C_2$.

$(\underline{A}\ f \in F \mid f \sqsubseteq h)$

$\Rightarrow$ {definition 3.9}

$(\underline{A}\ f \in F \mid (\underline{A}\ x \in C_1 \mid f(x) \sqsubseteq_2 h(x))$

$=$ {interchange quantifiers}

$(\underline{A}\ x \in C_1 \mid (\underline{A}\ f \in F \mid f(x) \sqsubseteq_2 h(x)))$

$\Rightarrow$ {definition lub in C_2}

$(\underline{A}\ x \in C_1 \mid \bigsqcup_{f \in F} f(x) \sqsubseteq_2 h(x))$

$=$ {definition g}

$(\underline{A}\ x \in C_1 \mid g(x) \sqsubseteq_2 h(x))$

$=$ {definition 3.9}

$g \sqsubseteq h$.

2. The proof that each finite subset of $C_1 \to C_2$ has a glb is a dual version of the proof above.

□

3.1.2. Strictness

Definition 3.12 {strictness}

1. Let $(C_1,\sqsubseteq_1)$ and $(C_2,\sqsubseteq_2)$ be cpo's (uccl's, dccl's, ccl's, cl's), and let $f \in C_1 \to C_2$.
   f is called $\perp$-strict if $f(\perp_{C_1}) = \perp_{C_2}$.

2. Let $(C_1,\sqsubseteq_1)$ and $(C_2,\sqsubseteq_2)$ be uccl's (dccl's, ccl's, cl's), and let $f \in C_1 \to C_2$.
   f is called $\top$-strict if $f(\top_{C_1}) = \top_{C_2}$.

□

For the determination of the strictness of a composite function in $C \to C$ the following theorem may be useful.

Theorem 3.13
Let $(C,\sqsubseteq)$ be a cpo.
Let $S_\perp = \{f \in C \to C \mid f \text{ is } \perp\text{-strict}\}$.

1. $(\lambda x \mid \perp) \in S_\perp$,
2. $(\lambda x \mid x) \in S_\perp$,
3. for all $f,g \in S_\perp$: $f \circ g \in S_\perp$.

If, moreover, $(C,\sqsubseteq)$ is a uccl, then

4. for each countable subset $F \subseteq S_\perp$: $(\lambda x \mid \bigsqcup_{f\in F} f(x)) \in S_\perp$,
5. for each finite nonempty subset $F \subseteq S_\perp$: $(\lambda x \mid \bigsqcap_{f\in F} f(x)) \in S_\perp$.

□

Proof
1,2,3: trivial.

4. $(\lambda x \mid \bigsqcup_{f\in F} f(x))(\perp_{C_1}) = \bigsqcup_{f\in F} f(\perp_{C_1}) = \bigsqcup_{f\in F} \perp_{C_2} = \perp_{C_2}$.

5. $(\lambda x \mid \bigsqcap_{f\in F} f(x))(\perp_{C_1}) = \bigsqcap_{f\in F} f(\perp_{C_1}) = \bigsqcap_{f\in F} \perp_{C_2} = \{F \text{ nonempty}\}\ \perp_{C_2}$.

3.1.3. Monotonicity

Definition 3.14 {monotonic function}
Let $(C_1,\sqsubseteq_1)$ and $(C_2,\sqsubseteq_2)$ be partially ordered sets. A function $f \in C_1 \to C_2$ is called monotonic if

$$(\underline{A}\ x,y \in C_1 \mid x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y)) \ .$$

The set of monotonic functions from C_1 to C_2 is denoted by $C_1 \to_m C_2$.
□

Lemma 3.15
Let $(C_1,\sqsubseteq_1)$ and $(C_2,\sqsubseteq_2)$ be partially ordered sets, and let $\sqsubseteq$ be the standard order on $C_1 \to C_2$.
If $(C_2,\sqsubseteq_2)$ is a cpo (uccl, dccl, ccl, cl), then $(C_1 \to_m C_2,\sqsubseteq)$ is a cpo (uccl, dccl, ccl, cl).
□

Proof
We only prove the upward case for uccl's; the other proofs are similar.
Let F be a countable subset of $C_1 \to_m C_2$.

$$\begin{aligned}
&(\underline{A}\ f \in F \mid (\underline{A}\ x,y \in C_1 \mid x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y)) \\
&= (\underline{A}\ x,y \in C_1 \mid (\underline{A}\ f \in F \mid x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y)) \\
&= (\underline{A}\ x,y \in C_1 \mid x \sqsubseteq_1 y \Rightarrow (\underline{A}\ f \in F \mid f(x) \sqsubseteq_2 f(y))) \\
&\Rightarrow (\underline{A}\ x,y \in C_1 \mid x \sqsubseteq_1 y \Rightarrow \bigsqcup_{f\in F} f(x) \sqsubseteq_2 \bigsqcup_{f\in F} f(y)) \\
&= (\underline{A}\ x,y \in C_1 \mid x \sqsubseteq_1 y \Rightarrow (\sqcup F)(x) \sqsubseteq_2 (\sqcup F)(y)) \\
&= \sqcup F \in C_1 \to C_2 \ .
\end{aligned}$$

□

3.1.4. Conjunctivity and disjunctivity

Definition 3.16 {$\sqcap$ and $\sqcup$ as infix operators}
Let $(C,\sqsubseteq)$ be a uccl.
The infix operators $\sqcap,\sqcup: C \times C \to C$ are defined by:

1. for all $x,y \in C$: $x \sqcap y = \sqcap \{x,y\}$,
2. for all $x,y \in C$: $x \sqcup y = \sqcup \{x,y\}$.

□

In terms of these operators we can define the notions conjunctivity and disjunctivity. In lattice theory these notions are usually called multiplicativity and additivity, respectively.

Definition 3.17 {conjunctivity, disjunctivity}
Let $(C_1,\sqsubseteq_1)$ and $(C_2,\sqsubseteq_2)$ be uccl's, and let $f \in C_1 \to C_2$.

1. f is called conjunctive if $(\underline{A}\ x,y \in C_1 \mid f(x \sqcap_1 y) = f(x) \sqcap_2 f(y))$
2. f is called disjunctive if $(\underline{A}\ x,y \in C_1 \mid f(x \sqcup_1 y) = f(x) \sqcup_2 f(y))$.

□

Lemma 3.18
Let $(C_1,\sqsubseteq_1)$ and $(C_2,\sqsubseteq_2)$ be uccl's, and let $f \in C_1 \to C_2$.

1. if f is conjunctive, then f is monotonic;
2. if f is disjunctive, then f is monotonic.

□

Proof
Let $x,y \in C_1$ such that $x \sqsubseteq_1 y$.

1. $f(x) = f(x \sqcap_1 y) = f(x) \sqcap_2 f(y) \sqsubseteq_2 f(y)$;
2. $f(y) = f(x \sqcup_1 y) = f(x) \sqcup_2 f(y) \sqsupseteq_2 f(x)$.

□

3.1.5. Continuity

Definition 3.19 {continuity}

1. Let $(C_1,\sqsubseteq_1)$ and $(C_2,\sqsubseteq_2)$ be cpo's.
   A monotonic function $f \in C_1 \to C_2$ is called upward continuous if for each ascending chain $\langle x_i \rangle_{i=0}^{\infty}$ in C_1:

   $$f(\bigsqcup_{i=0}^{\infty} x_i) = \bigsqcup_{i=0}^{\infty} f(x_i) .$$

   The set of upward continuous functions in $C_1 \to C_2$ is denoted by $C_1 \to_{uc} C_2$.

2. Let $(C_1,\sqsubseteq_1)$ and $(C_2,\sqsubseteq_2)$ be dccl's.
   A monotonic function $f \in C_1 \to C_2$ is called downward continuous if for each descending chain $\langle x_i \rangle_{i=0}^{\infty}$ in C_1:

   $$f(\bigsqcap_{i=0}^{\infty} x_i) = \bigsqcap_{i=0}^{\infty} f(x_i) .$$

   The set of downward continuous functions in $C_1 \to C_2$ is denoted by $C_1 \to_{dc} C_2$.

□

Note

In definition 3.19.1 it would be sufficient to require that $f(\bigsqcup_{i=0}^{\infty} x_i) \sqsubseteq_2 \bigsqcup_{i=0}^{\infty} f(x_i)$, because $f(\bigsqcup_{i=0}^{\infty} x_i) \sqsupseteq_2 \bigsqcup_{i=0}^{\infty} f(x_i)$ follows from the monotonicity of f. A complementary remark applies to definition 3.19.2.

□

Note

Since continuity is a stronger property then monotonicity it follows that all properties of monotonic functions also hold for continuous functions. In some of the forthcoming proofs use will be made of this fact without explicit reference.

□

As preparation for some important theorems concerning continuity we first present some lemmas.

Lemma 3.20

1. Let $(C,\sqsubseteq)$ be a cpo. Let, for $i \in \{0,1,\ldots\}$, $j \in \{0,1,\ldots\}$, x_{ij} be elements of C. If

$$(\underline{A}\ i,j,k,l \mid 0 \leq i \leq k \wedge 0 \leq j \leq l \mid x_{ij} \sqsubseteq x_{kl})$$

then

$$\bigsqcup_{i=0}^{\infty} \bigsqcup_{j=0}^{\infty} x_{ij} = \bigsqcup_{j=0}^{\infty} \bigsqcup_{i=0}^{\infty} x_{ij} = \bigsqcup_{k=0}^{\infty} x_{kk} .$$

2. Let $(C,\sqsubseteq)$ be a uccl. Let R,S be two countable sets. Let, for $i \in R$, $j \in S$, x_{ij} be elements of C. Then

$$\bigsqcup_{i\in R} \bigsqcup_{j\in S} x_{ij} = \bigsqcup_{j\in S} \bigsqcup_{i\in R} x_{ij} .$$

□

Proof

1. See [de Bakker].
2. See Appendix A.

□

Lemma 3.21

Let $\langle x_i \rangle_{i=0}^{\infty}$ and $\langle y_i \rangle_{i=0}^{\infty}$ be two ascending chains in a uccl $(C,\sqsubseteq)$.

1. $\langle x_i \sqcap y_i \rangle_{i=0}^{\infty}$ is an ascending chain.

2. $\bigsqcup_{i=0}^{\infty} (x_i \sqcap y_i) = (\bigsqcup_{i=0}^{\infty} x_i) \sqcap (\bigsqcup_{i=0}^{\infty} y_i)$.

□

Proof

See Appendix A.

□

Lemma 3.22

Let $(C,\sqsubseteq)$ be a uccl. Let S be a finite set. Let, for $i \in S$, $j \in \{0,1,\ldots\}$, x_{ij} be elements of C. If

$$(\underline{A}\ i,j,k \mid i \in S \wedge 0 \leq j \leq k \mid x_{ij} \sqsubseteq x_{ik})$$

then

$$\bigsqcup_{j=0}^{\infty} \bigsqcap_{i \in S} x_{ij} = \bigsqcap_{i \in S} \bigsqcup_{j=0}^{\infty} x_{ij}$$

□

Proof

See Appendix A.

□

Theorem 3.23

Let $(C_1, \sqsubseteq_1)$ and $(C_2, \sqsubseteq_2)$ be cpo's, and let $\sqsubseteq$ be the standard order on $C_1 \to C_2$.

1. - $(C_1 \to_{uc} C_2, \sqsubseteq)$ is a cpo;
   - $\bot_{C_1 \to_{uc} C_2} = (\lambda x \mid \bot_{C_2})$;
   - for each ascending chain $\langle f_i \rangle_{i=0}^{\infty}$ in $C_1 \to_{uc} C_2$:

$$\bigsqcup_{i=0}^{\infty} f_i = (\lambda x \mid \bigsqcup_{i=0}^{\infty} f_i(x)) \ .$$

2. If, moreover, $(C_2, \sqsubseteq_2)$ is a uccl, then
   - $(C_1 \to_{uc} C_2, \sqsubseteq)$ is a uccl;
   - for each countable $F \subseteq C_1 \to_{uc} C_2$: $\sqcup F = (\lambda x \mid \bigsqcup_{f \in F} f(x))$;
   - for each finite $F \subseteq C_1 \to_{uc} C_2$: $\sqcap F = (\lambda x \mid \bigsqcap_{f \in F} f(x))$.

□

Proof

1. See [de Bakker].
2. In view of theorem 3.10 it suffices to show that $(\lambda x \mid \bigsqcup_{f \in F} f(x))$ and $(\lambda x \mid \bigsqcap_{f \in F} f(x))$ are upward continuous.
   Let $\langle x_i \rangle_{i=0}^{\infty}$ be an ascending chain in C_1.

   1. $(\lambda x \mid \bigsqcup_{f \in F} f(x))(\bigsqcup_{i=0}^{\infty} x_i)$

      = {β-reduction} $\bigsqcup_{f \in F} f(\bigsqcup_{i=0}^{\infty} x_i)$

      = {$F \subseteq C_1 \to_{uc} C_2$} $\bigsqcup_{f \in F} \bigsqcup_{i=0}^{\infty} f(x_i)$

$$= \{\text{lemma 3.20.2}\}\ \bigsqcup_{i=0}^{\infty} \bigsqcup_{f\in F} f(x_i)$$

$$= \{\beta\text{-expansion}\}\ \bigsqcup_{i=0}^{\infty} (\lambda x \mid \bigsqcup_{f\in F} f(x))(x_i)\ .$$

2. The proof that $(\lambda x \mid \bigsqcap_{f\in F} f(x))$ is upward continuous is similar to the one given above, the main difference being that lemma 3.22 rather than lemma 3.20.2 has to be applied.

□

For the determination of the continuity of a composite function in $C \to C$ the following theorem may be useful:

Theorem 3.24

Let $(C,\sqsubseteq)$ be a uccl.

1. For all $y \in C$: $(\lambda x \mid y) \in C \to_{uc} C$;
2. $(\lambda x \mid x) \in C \to_{uc} C$;
3. for all $f,g \in C \to_{uc} C$: $f\circ g \in C \to_{uc} C$;
4. for each countable $F \subseteq C \to_{uc} C$: $(\lambda x \mid \bigsqcup_{f\in F} f(x)) \in C \to_{uc} C$;
5. for each finite $F \subseteq C \to_{uc} C$: $(\lambda x \mid \bigsqcap_{f\in F} f(x)) \in C \to_{uc} C$.

□

Proof

Let $\langle x_i\rangle_{i=0}^{\infty}$ be an ascending chain in C.

1. $(\lambda x \mid y)(\bigsqcup_{i=0}^{\infty} x_i) = y = \bigsqcup_{i=0}^{\infty} y = \bigsqcup_{i=0}^{\infty} (\lambda x \mid y)(x_i)$;

2. $(\lambda x \mid x)(\bigsqcup_{i=0}^{\infty} x_i) = \bigsqcup_{i=0}^{\infty} x_i = \bigsqcup_{i=0}^{\infty} (\lambda x \mid x)(x_i)$;

3. $(f\circ g)(\bigsqcup_{i=0}^{\infty} x_i) = f(g(\bigsqcup_{i=0}^{\infty} x_i)) = f(\bigsqcup_{i=0}^{\infty} g(x_i)) = \bigsqcup_{i=0}^{\infty} f(g(x_i))$
$= \bigsqcup_{i=0}^{\infty} (f\circ g)(x_i)$;

4, 5 immediately by theorem 3.23.2 and theorem 3.10.2.

□

Example

Let $(C,\sqsubseteq)$ be a uccl.

1. For i: $1 \leq i \leq n$: let $c_i \in C$;

2. for i: $1 \leq i \leq n$: let $d_i \in C$;

3. for i: $1 \leq i \leq n$: let $f_i \in C \to_{uc} C$.

We show that

$$(\lambda x \mid (\bigsqcup_{i=1}^{n} c_i) \sqcap \bigsqcap_{i=1}^{n} (d_i \sqcap f_i(x))) \in C \to_{uc} C \ .$$

4. {by 1, th. 3.24.1} for i: $1 \leq i \leq n$: $(\lambda x \mid c_i) \in C \to_{uc} C$;

5. {by 4, th. 3.24.4} $(\lambda x \mid \bigsqcup_{i=1}^{n} c_i) \in C \to_{uc} C$;

6. {by 2, th. 3.24.1} for i: $1 \leq i \leq n$: $(\lambda x \mid d_i) \in C \to_{uc} C$;

7. {by 3,6, th. 3.24.4} for i: $1 \leq i \leq n$: $(\lambda x \mid d_i \sqcup f_i(x)) \in C \to_{uc} C$;

8. {by 7, th. 3.24.5} $(\lambda x \mid \bigsqcap_{i=1}^{n} (d_i \sqcup f_i(x))) \in C \to_{uc} C$;

9. {by 5,8, th. 3.24.5} $(\lambda x \mid (\bigsqcup_{i=1}^{n} c_i) \sqcap \bigsqcap_{i=1}^{n} (d_i \sqcup f_i(x))) \in C \to_{uc} C$.

□

Note

In the sequel we will not give proofs like the one above in detail. Instead we will simply refer to "repeated application" of theorem 3.24.

□

The following lemma is used in the proof of theorem 3.26.

Lemma 3.25

Let $(C,\sqsubseteq)$ be a uccl. Let $D = C \to_{uc} C$.

1. If $g \in D$ and $\langle h_i \rangle_{i=0}^{\infty}$ is an ascending chain in D, then

$$g \circ \left(\bigsqcup_{i=0}^{\infty}{}^{D} h_i \right) = \bigsqcup_{i=0}^{\infty}{}^{D} (g \circ h_i) \ ;$$

2. If $\langle g_i \rangle_{i=0}^{\infty}$ and $\langle h_j \rangle_{j=0}^{\infty}$ are ascending chains in D, then

$$\left(\underline{|D|}_{i=0}^{\infty} g_i\right) \circ \left(\underline{|D|}_{j=0}^{\infty} h_j\right) = \underline{|D|}_{k=0}^{\infty} (g_k \circ h_k) \ .$$

□

Proof

See Appendix A.

□

Let $(C,\sqsubseteq_C)$ be a uccl, and let $D = C \to_{uc} C$. With $\sqsubseteq_D$ being the standard order on D we have, by theorem 3.23.2, that $(D,\sqsubseteq_D)$ is also a uccl. In the sequel we will often be concerned with functions $F \in D \to D$ which will usually be given in the form $F = (\lambda f \in D \mid E(f))$, where E(f) is an expression in terms of f, usually in the form $(\lambda x \in C \mid G(f,x))$. For the determination of the continuity of such a function from its composition the following theorem will be useful.

Theorem 3.26

Let $(C,\sqsubseteq_C)$ be a uccl, and let $D = C \to_{uc} C$.

1. For all $c \in C$: $(\lambda f \mid (\lambda x \mid c)) \in D \to_{uc} D$;
2. $(\lambda f \mid (\lambda x \mid x)) \in D \to_{uc} D$;
3. if $(\lambda f \mid E_1(f)) \in D \to_{uc} D$ and $(\lambda f \mid E_2(f)) \in D \to_{uc} D$, then
$$(\lambda f \mid E_1(f) \circ E_2(f)) \in D \to_{uc} D \ ;$$
4. if for all i: $1 \le i \le n$: $(\lambda f \mid (\lambda x \mid E_i)) \in D \to_{uc} D$, then
$$\left(\lambda f \mid (\lambda x \quad \underline{|C|}_{i=1}^{n} E_i)\right) \in D \to_{uc} D \ ;$$
5. if for all i: $1 \le i \le n$: $(\lambda f \mid (\lambda x \mid E_i)) \in D \to_{uc} D$, then
$$\left(\lambda f \mid (\lambda x \mid \overline{|C|}_{i=1}^{n} E_i)\right) \in D \to_{uc} D \ .$$

□

Proof

1. Let $c \in C$.

   $c \in C$

   $\Rightarrow$ {theorem 3.24.1, $D = C \rightarrow_{uc} C$}

   $(\lambda x \mid c) \in D$

   $\Rightarrow$ {theorem 3.24.1}

   $(\lambda f \mid (\lambda x \mid c)) \in D \rightarrow_{uc} D$.

2. true

   $\Rightarrow$ {theorem 3.24.2}

   $(\lambda x \mid x) \in D$

   $\Rightarrow$ {theorem 3.24.1}

   $(\lambda f \mid (\lambda x \mid x)) \in D \rightarrow_{uc} D$.

3. Let $\langle f_i \rangle_{i=0}^{\infty}$ be an ascending chain in D.

   $(\lambda f \mid E_1(f) \circ E_2(f))(\bigsqcup_{i=0}^{\infty} f_i)$

   = {β-reduction}

   $E_1(\bigsqcup_{i=0}^{\infty} f_i) \circ E_2(\bigsqcup_{i=0}^{\infty} f_i)$

   = {for i: $1 \leq i \leq 2$: $(\lambda f \mid E_i(f)) \in D \rightarrow_{uc} D$}

   $(\bigsqcup_{i=0}^{\infty} E_1(f_i)) \circ (\bigsqcup_{i=0}^{\infty} E_2(f_i))$

   = {lemma 3.25.2}

   $\bigsqcup_{i=0}^{\infty} E_1(f_i) \circ E_2(f_i)$

   = {β-expansion}

   $\bigsqcup_{i=0}^{\infty} (\lambda f \mid E_1(f) \circ E_2(f))(f_i)$.

4. $\left(\lambda f \mid (\lambda x \mid \underset{i=1}{\overset{n}{\lfloor\!\lfloor C \rfloor\!\rfloor}} E_i)\right)$

   = {theorem 3.23.2}

   $\left(\lambda f \mid \underset{i=1}{\overset{n}{\lfloor\!\lfloor D \rfloor\!\rfloor}} (\lambda x \mid E_i)\right)$

   = {theorem 3.23.2}

   $\underset{i=1}{\overset{n}{\lfloor\!\lfloor D \to_{uc} D \rfloor\!\rfloor}} (\lambda f \mid (\lambda x \mid E_i))$.

5. $\left(\lambda f \mid (\lambda x \mid \underset{i=1}{\overset{n}{\lceil\!\lceil C \rceil\!\rceil}} E_i)\right)$

   = {theorem 3.23.2}

   $\left(\lambda f \mid \underset{i=1}{\overset{n}{\lceil\!\lceil D \rceil\!\rceil}} (\lambda x \mid E_i)\right)$

   = {theorem 3.23.2}

   $\underset{i=1}{\overset{n}{\lceil\!\lceil D \to_{uc} D \rceil\!\rceil}} (\lambda f \mid (\lambda x \mid E_i))$.

□

3.1.6. Fixed points

Definition 3.27 {fixed point}
Let $(C,\sqsubseteq)$ be a cpo, $f \in C \to C$, $x \in C$

1. x is called a fixed point of f if $f(x) = x$.
2. x is called the least (greatest) fixed point of f if x is a fixed point of f and, moreover, for each fixed point y of f, $x \sqsubseteq y$ ($x \sqsupseteq y$).
   If f has a least (greatest) fixed point it is denoted by μf (νf).

□

Theorem 3.28

1. Let $(C,\sqsubseteq)$ be a cpo.
   If a function $f \in C \to C$ is upward continuous it has a least fixed

point satisfying

$$\mu f = \bigsqcup_{i=0}^{\infty} f^i(\perp_C) ,$$

where $f^0 = (\lambda x \mid x)$, $f^{i+1} = f \circ f^i$ for $i \geq 0$.

2. Let $(C,\sqsubseteq)$ be a dccl.
   If a function $f \in C \to C$ is downward continuous it has a greatest fixed point νf satisfying

$$\nu f = \bigsqcap_{i=0}^{\infty} f^i(\top_C) ,$$

where $f^0 = (\lambda x \mid x)$, $f^{i+1} = f \circ f_i$ for $i \geq 0$.

□

Proof

1. See [de Bakker].
2. By 1 and duality.

□

The following simple lemma will be used in several places in combination with the fixed point property $y = f(y)$. Together they form the basis for the inductive proof rules for repetitions and recursive procedures.

Lemma 3.29

Let $(C,\sqsubseteq)$ be a uccl. Let $\{x_i \mid i \geq 0\} \subseteq C$, $y \in C$. If

$$(\underline{A}\ i \mid i \geq 0 \mid (\bigsqcup_{0 \leq j < i} x_j \sqsubseteq y) \Rightarrow (x_i \sqsubseteq y))$$

then

$$\bigsqcup_{i=0}^{\infty} x_i \sqsubseteq y .$$

□

Proof

$$(\underline{A}\ i \mid i \geq 0 \mid (\bigsqcup_{0 \leq j < i} x_j \sqsubseteq y) \Rightarrow (x_i \sqsubseteq y))$$

= {definition lub}

$$(\underline{A}\ i \mid i \geq 0 \mid (\underline{A}\ j \mid 0 \leq j < i \mid x_j \sqsubseteq y) \Rightarrow (x_i \sqsubseteq y))$$

$= \{\text{math. ind.}\}$

$(\underline{A}\ i \mid i \geq 0 \mid x_i \sqsubseteq y)$

$= \{\text{definition lub}\}$

$\bigsqcup_{i=0}^{\infty} x_i \sqsubseteq y$

□

3.1.7. Fixed point induction

1. Let $(C,\sqsubseteq)$ be a cpo.
   In order to prove a property P of the least fixed point μf of an upward continuous function $f: C \to C$, the following induction rule may be used:

$$\frac{P(\bot),\ (\underline{A}\ x \mid P(x) \Rightarrow P(f(x)))}{P(\mu f)} .$$

2. Let $(C,\sqsubseteq)$ be a dccl.
   In order to prove a property P of the greatest fixed point νf of a downward continuous function $f: C \to C$, the following induction rule may be used:

$$\frac{P(\top),\ (\underline{A}\ x \mid P(x) \Rightarrow P(f(x)))}{P(\nu f)} .$$

In both cases it is necessary that the property P is "admissible".

Definition 3.30 {admissible predicate}

1. Let $(C,\sqsubseteq)$ be a cpo.
   A predicate P on C is called admissible for least fixed point induction if for all ascending chains $\langle x_i \rangle_{i=0}^{\infty}$:

$$(\underline{A}\ i \mid i \geq 0 \mid P(x_i)) \Rightarrow P(\bigsqcup_{i=0}^{\infty} x_i) .$$

2. Let $(C,\sqsubseteq)$ be a dccl.
   A predicate P on C is called admissible for greatest fixed point induction if for all descending chains $\langle x_i \rangle_{i=0}^{\infty}$:

$$(\underline{A}\ i \mid i \geq 0 \mid P(x_i)) \Rightarrow P(\bigsqcap_{i=0}^{\infty} x_i) .$$

□

The fixed point induction rule can easily be extended to systems of functions:

Let, for $n \geq 1$, $(C_1,\sqsubseteq_1),\ldots,(C_n,\sqsubseteq_n)$ be ccl's, and let $C = C_1 \times \ldots \times C_n$. On C we define the ordering $\sqsubseteq$ by:

$$(\underline{A}\ x_1,y_1 \in C_1,\ldots,x_n,y_n \in C_n \mid ((x_1,\ldots,x_n) \sqsubseteq (y_1,\ldots,y_n))$$

$$\Leftrightarrow (\underline{A}\ i \mid 1 \leq i \leq n \mid x_i \sqsubseteq_i y_i)) .$$

It can easily be verified that $(C,\sqsubseteq)$ is also a ccl.

Let k: $0 \leq k \leq n$.
Let for i: $1 \leq i \leq k$: $f_i \in C \to_{uc} C_i$.
Let for i: $k+1 \leq i \leq n$: $f_i \in C \to_{dc} C_i$.

In order to prove a property of the least fixed points $\mu f_1,\ldots,\mu f_k$ and the greatest fixed points $\nu f_{k+1},\ldots,\nu f_n$ the following induction rule may be used:

$$\frac{P((\bot_1,\ldots,\bot_k,\top_{k+1},\ldots,\top_n)),\quad (\underline{A}\ \bar{x} \in C \mid P(\bar{x}) \Rightarrow P((f_1(\bar{x}),\ldots,f_n(\bar{x}))))}{P((\mu f_1,\ldots,\mu f_k,\nu f_{k+1},\ldots,\nu f_n))} .$$

We will refer to this version of fixed point induction as <u>simultaneous fixed point induction</u>

The notion of admissibility has to be extended accordingly: P is <u>admissible for simultaneous fixed point induction</u> if

for all i: $1 \leq i \leq k$: for all ascending chains $\langle x_{ij}\rangle_{j=0}^{\infty}$ in C_i and
for all i: $k+1 \leq i \leq n$: for all descending chains $\langle x_{ij}\rangle_{j=0}^{\infty}$ in C_i:

$$(\underline{A}\ j \mid j \geq 0 \mid P(x_{1j},\ldots,x_{kj},x_{(k+1,j)},\ldots,x_n))$$

$$\Rightarrow P(\bigsqcup_{j=0}^{\infty} x_{1j},\ldots,\bigsqcup_{j=0}^{\infty} x_{kj},\bigsqcap_{j=0}^{\infty} x_{(k+1,j)},\ldots,\bigsqcap_{j=0}^{\infty} x_{nj}) .$$

3.2. The condition transformers wp and wlp

3.2.0. Introduction

In this section we will develop a condition transformer semantics for the kernel language. The form of this semantics is determined by two requirements.

- For reasons already mentioned in section 3.0, we want to consider conditions as elements of a formal language.
- For a proper treatment of recursion we want to avail of a suitable lattice structure on sets of conditions, sets of condition transformers, etc.

The design of a semantics with these properties requires some care. As starting point for such a design let us consider the attribute grammar for the kernel language presented in section 2.3.2, with the exception of grammar rule 13: nested blocks will be dealt with in chapter 4. We recall from section 2.3.2 that for an attributed nonterminal Stat <e> the attribute e ∈ *Env* determines the collection of variables, together with their types, that may occur in elements of L(Stat <e>). A good candidate for a corresponding condition language seems to be the set of first-order formulae over the same variables and over the operators of the kernel language. Unfortunately that set does not possess a suitable lattice structure. As the partial order should correspond to the implication, the least upper bound of a set of conditions would correspond to the disjunction of those conditions. In general such a disjunction cannot be represented by an element of the aforementioned condition language, however. Consider e.g. the set of conditions {n = 1, n = 1*2, n = 1*2*3,...}. The least upper bound of this chain would be the infinite disjunction

$$n = 1 \vee n = 1*2 \vee n = 1*2*3 \vee \ldots .$$

First order formulae equivalent to this disjunction would be

$$(\underline{E}\ i \mid i \geq 1 \mid n = i!)$$

or

$$(\underline{E}\ i \mid i \geq 1 \mid n = (\underline{P}\ j \mid 1 \leq j \wedge j \leq i \mid j))$$

but both contain operators not present in the kernel language itself.

Extension of the first order language with additional operators raises the problem of determining whether the extended language is closed under infinite disjunction or conjunction. A simpler way out was shown by Back [Back 1, Back 2], who proposed to allow infinite disjunctions and conjunctions as elements of the condition language. This gives rise to formulae of the so-called infinitary logic $L_{\omega_1\omega}$. This logic is much like ordinary first-order logic, but in addition allows for disjunctions and conjunctions over countable sets of formulae, and for proof rules to handle these formulae. In [Back 1] it is shown that the logic $L_{\omega_1\omega}$ is sufficiently rich to express the conditions required for the guarded commands. For increasingly extensive discussions of $L_{\omega_1\omega}$ we refer to [Back 1], [Back 2], [Scott 1] and [Karp], respectively.

From $L_{\omega_1\omega}$ it is not difficult to obtain a set of conditions that has a ccl structure. In section 3.2.1 we will define the condition language corresponding to an attribute e $\in$ *Env*. In section 3.2.2 we mention the essential features of a logic D capable of handling such conditions. In section 3.2.3 we define a ccl structure for conditions and condition transformers. In section 3.2.4 we reconsider the conditions transformers wp and wlp of [Dijkstra 2] in this framework.

3.2.1. Conditions

The condition language corresponding to an attribute e $\in$ *Env* can be defined by means of an attribute grammar which is very similar to that of the kernel language itself. The main difference is in the description of infinite formulae, which requires some extensions to the grammatical tools of chapter 2. Here we will only give a short sketch of these extensions. Their feasibility follows from the definitions of substitution and concatenation for infinite sequences given in [Karp]. We only mention the extensions for context-free grammars; those for attribute grammars are similar.

- First, we introduce production rules with an infinite right-hand side. These will be given in the form A ::= ∞ α ■, where α is an infinite sequence of nonterminals and terminals.
- Second, we extend the definition 2.2 of the relations >> and *>> to infinite sequences. Let w_1 and w_2 be two finite or infinite

sequences; then w_1 >> w_2 holds fif w_2 can be obtained from w_1 by replacing each occurrence A_i of a nonterminal A by a finite or infinite sequence α_i such that A ::= α_i ■ or A ::= ∞ α_i ■ is a production rule. The relation *>> is the reflexive and transitive closure of >>. (Note that in this way a derivation consists of a finite number of steps, each of which may involve an infinite number of substitutions. In terms of derivation trees this corresponds to trees of finite height, the nodes of which may have an infinite number of branches.)

- Third, for each (finite or infinite) sequence w_1 the set $L(w_1)$ is the set of all (finite or infinite) sequences w_2 of terminal symbols such that w_1 *>> w_2.

With the extensions sketched above it is not hard to define an infinitary condition language. As already said the attribute e ∈ *Env* determines the context of statements in L(Stat <e>), i.e. the set of admissible variables with their types. It is our intention that the corresponding condition language consists of infinitary first-order formulae over the same variables. In principle we could define such a condition language by means of a second attribute grammar, but as this grammar has much in common with that of the kernel language itself it will be easier to extend the latter grammar with some nonterminals, terminals and grammar rules. The extensions are as follows:

Nonterminals

{Cond <*Env*>, Cexpr <*Env*,*Prio*,*Type*>, Conj <*Env*>, Disj <*Env*>, Quant} .

Terminals

{"A","E"} .

Grammar rules

CL1. Cond <e> ::= Cexpr <e,p,t> ■
t = bool

Cl2. Cexpr <e,p_0,t_0> ::= Cexpr <e,p_1,t_1> Dop <p_0,t_0,t_1,t_2>
Cexpr <e,p_2,t_2> ■
$p_0 \leq p_1$
$p_0 < p_2$

CL3. Cexpr <e,p_0,t_0> ::= Mop <t_0,t_1> Cexpr <e,p_1,t_1> ■
$p_0 = 7$
$p_1 = 7$

CL4. Cexpr <e,p_0,t> ::= (Cexpr <e,p_1,t>) ■
$p_0 = 7$

CL5. Cexpr <e_0,p_0,t> ::= (Quant Decs <d> | Cexpr <e_1,p_1,t>
| Cexpr <e_1,p_2,t> ■
$t = bool$
$(\underline{A}\ n: Name \mid \#_D(n,d) \leq 1)$
$e_1 = Ext(e_0,d)$
$p_0 = 7$

CL6. Cexpr <e,p,t> ::= Var <e,n,t> ■
$p = 7$

CL7. Cexpr <e,p,t> ::= Con <t> ■
$p = 7$

CL8. Cexpr <e,p,t> ::= Disj <e> ■
$p = 2$
$t = bool$

CL9. Cexpr <e,p,t> ::= Conj <e> ■
$p = 3$
$t = bool$

CL10. Let α be such that $dom(\alpha) = \mathbb{N}$ and for all i: $0 \leq i$:
$\alpha_{2i} = Cexpr\ \langle e,p,t\rangle$, $\alpha_{2i+1} = \vee$

Disj <e> ::=$^{\infty}$ α ■
$p > 2$
$t = bool$

CL11. Let α be such that $dom(\alpha) = \mathbb{N}$ and for all i: $0 \leq i$:
$\alpha_{2i} = Cexpr\ \langle e,p,t\rangle$, $\alpha_{2i+1} = \wedge$.

Conj <e> ::=$^{\infty}$ α ■
$p > 3$
$t = bool$

CL12. Quant ::= A □

CL13. Quant ::= E □

The condition language that may be used with L(Stat <e>) is now simply defined as L(Cond <e>). By induction it is not hard to prove that for all e ∈ *Env*, p ∈ *Prio*:

L(Expr <e,p,bool>) ⊆ L(Cond <e>) .

Example

Let d ∈ *Decs*, e = Ext(Empty,d) be such that

```
Prog
*>>
|[ var Decs <d> | Stat <e> ]|
*>>
|[ var x,y: int,b: bool | Stat <e> ]| .
```

The set L(Stat <e>) contains elements like

```
skip
b := x > 3
x,y := 0,0; do x ≠ 10 → x := x + 1; y := y + x * x od
```

but not

```
c := 3
b := x + 1
if x → skip fi .
```

The set L(Cond <e>) contains elements like

```
true
b ∨ x > y
(A z: int | z < x | z < y)
```

but not

x
b > 3
(A a: bool | true | a > 3) .

□

Note
In the sequel we will usually abbreviate conditions of the form (A x: t | true | q) or (E x: t | true | q) to (A x: t ‖ q) and (A x: t ‖ q), respectively.

□

Note
We will often have to reason about conditions that are conjunctions or disjunctions of the elements of a countable set $Q \subseteq L(\text{Cond } \langle e \rangle)$. We will denote such conditions by $\bigwedge Q$ and $\bigvee Q$ respectively.
In case Q is a finite set such as $\{q_i \in L(\text{Cond } \langle e \rangle) \mid 1 \leq i \leq n\}$ we will also write $[\bigwedge i \mid 1 \leq i \leq n \mid q_i]$ and $[\bigvee i \mid 1 \leq i \leq n \mid q_i]$ respectively.
In case Q is an infinite set such as $\{q_i \in L(\text{Cond } \langle e \rangle) \mid 1 \leq i\}$ we will also write $[\bigwedge i \mid 1 \leq i \mid q_i]$ and $[\bigvee i \mid 1 \leq i \mid q_i]$, respectively.
It should be borne in mind that these notations are just abbreviations and do not themselves belong to the condition language!

□

3.2.2. The logic D

Calculations with conditions will be based upon a logic called D. The logic D differs from ordinary first-order predicate logic in two respects. First, like $L_{\omega_1\omega}$ it allows for infinitary formulae and proof rules to handle these. Second, it reflects that conditions are only meaningful relative to a certain context. The formulae of D are of the form c|p, where c is a sequence of declaration parts corresponding to an attribute e ∈ *Env* and p is an element of $L(\text{Cond } \langle e \rangle)$. More precisely, the set of formulae is the set $L(\text{Form } \langle Env \rangle)$ defined by the extensions below to the attribute grammar for the condition language. As preparation for manipulations with contexts in theorem 3.54 and in chapter 4 these extensions also contain two operations, new and rep, on environments. new(x,e) indicates that a name x does not occur in an environ-

ment e. rep(e,e',x,x') indicates that e' differs from e only in that each occurrence of x is replaced by x'. It will only be used with x' such that new(x',e) holds. The extensions follow.

Operations on *Env*

new(•,•) : *Name* * *Env* → *Bool*
rep(•,•,•,•): *Env* * *Env* * *Name* * *Name* → *Bool*

new(x,e) = ¬ (E t: *Type* | (x,t) $\underline{in}_E$ e)

rep(e,e',x,x') = (A y: *Name*,t: *Type* |
y ≠ x ∧ y ≠ x' | (y,t) $\underline{in}_E$ e ⇔ (y,t) $\underline{in}_E$ e')
∧ (A t: *Type* | (x,t) $\underline{in}_E$ e ⇔ (x',t) $\underline{in}_E$ e') .

Nonterminals

{Form <*Env*>, Cont <*Env*>}

Terminals

{"▷"}

Grammar rules

Form <e> ::= Cont <e> | Cond <e> ■

Cont <e> ::= ■
e = Empty

Cont <e_0> ::= Cont <e_1> > Decs <d> ■
e_0 = Ext(e_1,d)

Examples

▷ x,y: int ▷ b: bool | b ⇒ x > y

▷ x,y: int | (A b: bool ‖ b ⇒ x > y)
| (A x,y: int ‖ (A b: bool ‖ b ⇒ x > y))

▷ x,y: int | x > y
□

For formulae of the form $c \triangleright d \mid p$ the order and grouping of variables in d is irrelevant; e.g. the formula

$$c \triangleright x_1,\ldots,x_m\colon t_1,\ y_1,\ldots,y_n\colon t_2 \mid p$$

is equivalent to the formula

$$c \triangleright y_1\colon t_2,\ldots,y_n\colon t_2,\ x_m\colon t_1,\ldots,x_1\colon t_1 \mid p\ .$$

If a declaration x: t occurs in the part d of a formula $c \triangleright d \mid p$, then within p x stands for an arbitrary value of type t. This is reflected in the axiom that $c \triangleright \ldots,x\colon t \mid p$ is equivalent to $c \triangleright \ldots \mid (A\ x\colon t \parallel p)$. In particular the example formulae are all equivalent.

The proof rules of D are of the form

$$\frac{c_0 \mid p_0,p_1,\ldots}{c_1 \mid p}$$

and may contain a countable number of premises.

Most proof rules have direct counterparts in first-order predicate logic. We will only mention the axioms and proof rules for infinitary formulae:

Let $\{q_j \mid 0 \leq j\}$ be a countable set of conditions in a context c

DA1. For all j: $0 \leq j$: $c \mid [\bigwedge i \mid 0 \leq i \mid q_i] \Rightarrow q_j$

DA2. For all j: $0 \leq j$: $c \mid q_j \Rightarrow [\bigvee i \mid 0 \leq i \mid q_i]$

DR1. $$\frac{c \mid p \Rightarrow q_0,\ p \Rightarrow q_1,\ \ldots}{c \mid p \Rightarrow [\bigwedge i \mid 0 \leq i \mid q_i]}$$

Dr2. $$\frac{c \mid q_0 \Rightarrow p,\ q_1 \Rightarrow p,\ \ldots}{c \mid [\bigvee i \mid 0 \leq i \mid q_i] \Rightarrow p}$$

Provability of a formula $c \mid p$ in D will be indicated by: $\vdash_D c \mid p$.

3.2.3. The ccl's of conditions and condition transformers

In this section we will impose a suitable lattice structure on conditions and condition transformers, respectively. First we will partition a condition language into equivalence classes:

Definition 3.31 {$\underline{eq}$}
For all $e \in \mathit{Env}$ the condition $\underline{eq}_e$ on $L(\text{Cond } \langle e \rangle)$ is defined by:
for all $p,q \in L(\text{Cond } \langle e \rangle)$, $c \in L(\text{Cont } \langle e \rangle)$:

$$p \ \underline{eq}_e \ q \quad \text{fif} \quad \vdash_D c \mid p \Leftrightarrow q \ .$$

□

Example
Let $e \in \mathit{Env}$ be such that $\vdash (x,\text{int}) \ \underline{in}_E \ e$ and $\vdash (y,\text{int}) \ \underline{in}_E \ e$.

$$x > y \quad \underline{eq}_e \ \neg (x \leq y)$$

$$x = x+1 \ \underline{eq}_e \ \text{false}$$

$$x > y \quad \underline{eq}_e \ (\underline{A} \ z\text{: int} \mid z = x \mid z > y) \ .$$

□

Definition 3.32 {C_e}
For all $e \in \mathit{Env}$: $C_e = L(\text{Cond } \langle e \rangle) \ / \ \underline{eq}_e$.
□

The partitioning into equivalence classes serves to get rid of the complications stemming from the fact that different conditions may characterize the same set of states (viz. when $p \neq q$, but $p \ \underline{eq}_e \ q$). Henceforth we will identify C_e with $L(\text{Cond } \langle e \rangle)$ and freely replace conditions by equivalent ones.

Definition 3.33 {$\underline{\sqsubseteq}_e$}
For all $e \in \mathit{Env}$, the relation $\underline{\sqsubseteq}_e$ on C_e is defined by:
for all $p,q \in C_e$, $c \in L(\text{Cont } \langle e \rangle)$:

$$p \ \underline{\sqsubseteq}_e \ q \quad \text{fif} \quad \vdash_D c \mid p \Rightarrow q \ .$$

□

Example

Let e be as in the previous example.

$$x > y \sqsubseteq_e x > y-1$$

$$x > y \sqsubseteq_e (\underline{A}\ z\colon \text{int} \mid y > z \mid x > z)$$

$$\text{false} \sqsubseteq_e x > y\ .$$

□

It can easily be verified that $\sqsubseteq_e$ is a partial order on C_e. In fact:

Theorem 3.34

For all $e \in Env$: $(C_e, \sqsubseteq_e)$ is a ccl.

□

Proof

Let Q be a countable subset of C_e.

$\sqcup Q = \bigvee Q$, on account of definition 3.33, axiom DA_2 and rule DR2.

$\sqcap Q = \bigwedge Q$, on account of definition 3.33, axiom DA1 and rule DR1.

□

Note that, in particular:

$$\bot_{C_e} = \sqcup \emptyset = \text{false}$$

$$\top_{C_e} = \sqcap \emptyset = \text{true}.$$

Next we will impose a lattice structure on condition transformers.

Definition 3.35 $\{T_e\}$

For all $e \in Env$:

$$T_e = C_e \to C_e\ .$$

□

Theorem 3.36

Let $e \in Env$. Let $\sqsubseteq_{T_e}$ be the standard order on T_e.
$(T_e, \sqsubseteq_{T_e})$ is a ccl.

□

Proof

Immediately by theorem 3.10.2.

□

3.2.4. Definitions and some properties of wp and wlp

The fact that for any e $\in$ *Env* both C_e and T_e are ccl's enables us to discuss the condition transformers wp and wlp in a lattice-theoretical framework. We begin with presenting our versions of the definitions of wp and wlp.

Definition 3.37 {wp}
For all e $\in$ *Env* the function $wp_e \in L(\text{Stat } \langle e \rangle) \to T_e$ is defined by:

1. $wp_e(\text{abort}) = (\lambda q \in C_e \mid \text{false})$
2. $wp_e(\text{skip}) = (\lambda q \in C_e \mid q)$
3. $wp_e(v := E) = (v \leftarrow E)$ {see first note below}
4. $wp_e(S_1;S_2) = wp_e(S_1) \circ wp_e(S_2)$
5. $wp_e(\underline{\text{if}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{fi}}) =$

   $(\lambda q \in C_e \mid [\bigvee i \mid 1 \le i \le n \mid B_i] \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)(q)])$
6. $wp_e(\underline{\text{do}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{od}}) = \mu F$,

   where $F = (\lambda f \in T_e \mid (\lambda q \in C_e \mid ([\bigvee i \mid 1 \le i \le n \mid B_i] \vee q)$

   $\wedge\ [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)(f(q))]))$.

Definition 3.38 {wlp}
For all e $\in$ *Env* the function $wlp_e \in L(\text{Stat } \langle e \rangle) \to T_e$ is defined by:

1. $wlp_e(\text{abort}) = (\lambda q \in C_e \mid \text{true})$
2. $wlp_e(\text{skip}) = (\lambda q \in C_e \mid q)$
3. $wlp_e(v := E) = (v \leftarrow E)$ {see first note below}
4. $wlp_e(S_1;S_2) = wlp_e(S_1) \circ wlp_e(S_2)$
5. $wlp_e(\underline{\text{if}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{fi}}) =$

   $(\lambda q \in C_e \mid [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow wlp_e(S_i)(q)])$

6. $wlp_e(\underline{do}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{od}) = \nu G$,

where $G = (\lambda f \in T_e \mid (\lambda q \in C_e \mid ([\bigvee i \mid 1 \le i \le n \mid B_i] \vee q) \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow wlp_e(S_i)(f(q))]))$.

□

Note

The operator $(v \leftarrow E)$ occurring in definitions 3.37.3 and 3.38.3 is an instance of the substitution operator for conditions, that we assume to have been defined in the customary way with precautions to avoid name clashes. See e.g. [Curry, de Bakker].

□

Note

In the sequel we will often omit the brackets around the condition argument of functions like wp and wlp, writing e.g. $wp_e(S_1)wp_e(S_2)q$ instead of $wp_e(S_1)(wp_e(S_2)(q))$.

□

In definition 3.37.6 $wp_e(\underline{do}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{od})$ is well-defined only if the least fixed point μF of F indeed exists. According to theorem 3.28.1 existence of μF is guaranteed if F is upward continuous, which in its turn depends on upward continuity (and well-definedness!) of $wp_e(S_i)$ for i: $1 \le i \le n$. A similar remark applies to downward continuity of G in definition 3.38.6.

Theorem 3.39 simultaneously states the continuity of $wp_e(S)$ and F (and also that of $wlp_e(S)$ and G), thereby justifying the definitions above. This theorem is the first of a number of theorems that express some important properties of wp and wlp. Some of these are reformulations of properties mentioned in [Dijkstra 2]. Most of the proofs make use of both structural induction on the composition of statements and fixed point induction. Induction hypotheses are indicated by the letter H and an index.

Theorem 3.39

For all $e \in \mathit{Env}$, $S \in L(\text{Stat } \langle e \rangle)$:

1. $wp_e(S) \in C_e \rightarrow_{uc} C_e$.

   If S is of the form $\underline{\text{do}}\ B_1 \rightarrow S_1 \ [] \ \ldots \ [] \ B_n \rightarrow S_n \ \underline{\text{od}}$, and F is as in definition 3.37.6, then $F \in (C_e \rightarrow_{uc} C_e) \rightarrow_{uc} (C_e \rightarrow_{uc} C_e)$.

2. $wlp_e(S) \in C_e \rightarrow_{dc} C_e$.

   If S is of the form $\underline{\text{do}}\ B_1 \rightarrow S_1 \ [] \ \ldots \ [] \ B_n \rightarrow S_n \ \underline{\text{od}}$, and G is as in definition 3.38.6, then $G \in (C_e \rightarrow_{dc} C_e) \rightarrow_{dc} (C_e \rightarrow_{dc} C_e)$.

□

Proof

We only prove 1. The proof of 2 is similar when dual versions of theorems 3.24 and 3.26 are used.

The proof is by induction on the composition of S.

1.1. S :: abort

Immediately by theorem 3.24.1.

1.2. S :: skip

Immediately by theorem 3.24.2.

1.3. S :: v := E

Let $\langle q_i \rangle_{i=0}^{\infty}$ be an ascending chain in C_e.

$wp_e(v := E)[\bigvee i \mid 0 \leq i \mid q_i]$

= {definition 3.37.3}

$(v \leftarrow E)[\bigvee i \mid 0 \leq i \mid q_i]$

= $[\bigvee i \mid 0 \leq i \mid (v \leftarrow E)q_i]$

= {definition 3.37.3}

$[\bigvee i \mid i \leq 0 \mid wp_e(v \leftarrow E)q_i]$.

1.4. S :: $S_1;S_2$

H1. $wp_e(S_1) \in C_e \rightarrow_{uc} C_e$.

H2. $wp_e(S_2) \in C_e \rightarrow_{uc} C_e$.

Immediately by theorem 3.24.3.

1.5. $S :: \underline{\text{if}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{fi}}$

H3. For all i: $1 \leq i \leq n$: $wp_e(S_i) \in C_e \to_{uc} C_e$.

$wp_e(S)$

= {definition 3.37.5}

$(\lambda q \in C_e \mid [\bigvee i \mid 1 \leq i \leq n \mid B_i] \wedge [\bigwedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow wp_e(S_i)q])$

= {prop. log.}

$(\lambda q \in C_e \mid [\bigvee i \mid 1 \leq i \leq n \mid B_i] \wedge [\bigwedge i \mid 1 \leq i \leq n \mid \neg B_i \vee wp_e(S_i)q])$.

From H3 and repeated application of theorem 3.24 it follows that $wp_e(S) \in C_e \to_{uc} C_e$.

1.6. $S :: \underline{\text{do}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{od}}$

H4. For all i: $1 \leq i \leq n$: $wp_e(S_i) \in C_e \to_{uc} C_e$.

From H4 and repeated application of theorem 3.26 it follows that $F \in (C_e \to_{uc} C_e) \to_{uc} (C_e \to_{uc} C_e)$.

Hence μF exists and is an element of $C_e \to_{uc} C_e$, so $wp_e(S) \in C_e \to_{uc} C_e$.

□

Corollary 3.40

For all $e \in Env$, $S \in L(\text{Stat } \langle e \rangle)$:

1. $wp_e(S) \in C_e \to_m C_e$.
2. $wlp_e(S) \in C_e \to_m C_e$.

□

Theorem 3.41

For all $e \in Env$, $S \in L(\text{Stat } \langle e \rangle)$:

1. $wp_e(S)$ is $\perp$-strict.
2. $wlp_e(S)$ is $\top$-strict.

□

Proof

We only prove 1; the proof of 2 is similar. The proof is by induction on the composition of S.

1.1. S :: abort

Immediately by theorem 3.13.1.

1.2. S :: skip

Immediately by theorem 3.13.2.

1.3. S :: v := E

wp(v := E) false

= {definition 3.37.3} (v ← E) false

= false.

1.4. S :: $S_1;S_2$

H1. $wp_e(S_1)$ is $\perp$-strict.

H2: $wp_e(S_2)$ is $\perp$-strict.

$wp(S_1;S_2)$ false

= {definition 3.37.4} $wp(S_1)wp(S_2)$ false

= {H2} $wp(S_1)$ false

= {H1} false.

1.5. S :: if $B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n$ fi

H3. For all i: $1 \le i \le n$: $wp_e(S_i)$ is $\perp$-strict.

wp(if $B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n$ fi) false

= {definition 3.37.5}

$[\bigvee i \mid 1 \le i \le n \mid B_i] \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)\ \text{false}]$

= {H3} $[\bigvee i \mid 1 \le i \le n \mid B_i] \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow \text{false}]$

= {prop. log.} false.

1.6. S :: do $B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n$ od

H4. For all i: $1 \le i \le n$: $wp_e(S_i)$ is $\perp$-strict.

The proof that μF is $\perp$-strict is by least fixed point induction. Admissibility is trivial.

1.6.1. {base step}

$$\begin{aligned} & \perp_{T_e} \text{ (false)} \\ = \ & (\lambda q \in C_e \mid \text{false}) \text{ false} \\ = \ & \text{false}. \end{aligned}$$

1.6.2. {induction step}

H5. f is $\perp$-strict.

$$\begin{aligned} & F(f) \text{ false} \\ = \ & \{\text{definition 3.37.6}\} \\ & ([\bigvee i \mid 1 \le i \le n \mid B_i] \vee \text{false}) \\ & \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)f(\text{false})] \\ = \ & \{\text{H5,H4}\} \\ & ([\bigvee i \mid 1 \le i \le n \mid B_i] \vee \text{false}) \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow \text{false}] \\ = \ & \{\text{prop. log.}\} \text{ false}. \end{aligned}$$

□

Theorem 3.42

For all $e \in \mathit{Env}$, $S \in L(\text{Stat } \langle e \rangle)$:

1. $wp_e(S)$ is conjunctive.
2. $wlp_e(S)$ is conjunctive.

□

Proof

We only prove 1. The proof is by induction on the composition of S.

Let $p,q \in C_e$.

1.1. S :: abort

$$\begin{aligned} & wp_e(\text{abort})(p \wedge q) \\ = \ & \{\text{definition 3.37.1}\} \text{ false} \\ = \ & \text{false} \wedge \text{false} \\ = \ & \{\text{definition 3.37.1 twice}\} \ wp_e(\text{abort})p \wedge wp_e(\text{abort})q \ . \end{aligned}$$

1.2. S :: skip

$wp_e(\text{skip})(p \wedge q)$

$= \{\text{definition 3.37.2}\}\ p \wedge q$

$= \{\text{definition 3.37.2 twice}\}\ wp_e(\text{skip})p \wedge wp_e(\text{skip})q$.

1.3. S :: v := E

$wp_e(v := E)(p \wedge q)$

$= \{\text{definition 3.37.3}\}\ (v \leftarrow E)(p \wedge q)$

$= (v \leftarrow E)p \wedge (v \leftarrow E)q$

$= \{\text{definition 3.37.3 twice}\}\ wp_e(v := E)p \wedge wp_e(v := E)q$.

1.4. S :: $S_1;S_2$

H1. $wp_e(S_1)$ is conjunctive.

H2. $wp_e(S_2)$ is conjunctive.

$wp(S_1;S_2)(p \wedge q)$

$= \{\text{definition 3.37.4}\}\ wp_e(S_1)wp_e(S_2)(p \wedge q)$

$= \{\text{H2}\}\ wp_e(S_1)(wp_e(S_2)p \wedge wp_e(S_2)q)$

$= \{\text{H1}\}\ wp_e(S_1)wp_e(S_2)p \wedge wp_e(S_1)wp_e(S_2)q$

$= \{\text{definition 3.37.4 twice}\}\ wp(S_1;S_2)p \wedge wp(S_1;S_2)q$.

1.5. S :: if $B_1 \rightarrow S_1 \ [] \ \ldots \ \rightarrow B_n \ [] \ S_n$ fi

H3. For all i: $1 \leq i \leq n$: $wp_e(S_i)$ is conjunctive.

$wp_e(S)(p \wedge q)$

$= \{\text{definition 3.37.5}\}$

$[\bigvee i \mid 1 \leq i \leq n \mid B_i] \wedge [\bigwedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow wp_e(S_i)(p \wedge q)]$

$= \{\text{H3}\}$

$[\bigvee i \mid 1 \leq i \leq n \mid B_i] \wedge [\bigwedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow (wp_e(S_i)p \wedge wp_e(S_i)q)]$

$= \{\text{prop. log.}\}$

$[\bigvee i \mid 1 \leq i \leq n \mid B_i] \wedge [\bigwedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow wp_e(S_i)p]$

$\wedge\ [\vee i \mid 1 \le i \le n \mid B_i] \wedge [\wedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)q]$

$= \{\text{definition 3.37.5 twice}\}\ wp_e(S)p \wedge wp_e(S)q$.

1.6. $S :: \underline{\text{do}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{od}}$

H4. For all i: $1 \le i \le n$: $wp_e(S_i)$ is conjunctive.

The proof that μF is conjunctive is by least fixed point induction. Admissibility follows from lemma 3.21.

1.6.1. {base step}

See 1.1.

1.6.2. {induction step}

H5. f is conjunctive.

$F(f)(p \wedge q)$

$= \{\text{definition 3.37.6}\}$

$([\vee i \mid 1 \le i \le n \mid B_i] \vee (p \wedge q))$

$\wedge\ [\wedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)f(p \wedge q)]$

$= \{H5, H4\}$

$([\vee i \mid 1 \le i \le n \mid B_i] \vee (p \wedge q))$

$\wedge\ [\wedge i \mid 1 \le i \le n \mid B_i \Rightarrow (wp_e(S_i)fp \wedge wp_e(S_i)fq)]$

$= \{\text{prop. log.}\}$

$([\vee i \mid 1 \le i \le n \mid B_i] \vee p) \wedge [\wedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)fp]$

$\wedge\ ([\vee i \mid 1 \le i \le n \mid B_i] \vee q) \wedge [\wedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)fq]$

$= \{\text{definition 3.37.6 twice}\}\ F(f)p \wedge F(f)q$.

□

Theorem 3.43

For all $e \in \mathit{Env}$, $S \in L(\text{Stat } \langle e \rangle)$, $q \in C_e$:

$$wp_e(S)q = wp_e(S)\text{true} \wedge wlp_e(S)q \ .$$

□

Proof

The proof is by induction on the composition of S. Let $q \in C_e$.

1. S :: abort

$wp_e(abort)q$

= {definition 3.37.1} false

= {prop. log.} false $\wedge$ true

= {definitions 3.37.1, 3.38.1} $wp_e(abort)true \wedge wlp_e(abort)q$.

2. S :: skip

$wp_e(skip)q$

= {definition 3.37.2} q

= {prop. log.} true $\wedge$ q

= {definitions 3.37.2, 3.38.2} $wp_e(skip)true \wedge wlp_e(skip)q$.

3. S :: v := E

$wp_e(v := E)q$

= {definition 3.37.3} $(v \leftarrow E)q$

= {subst. prop. log.} $(v \leftarrow E)true \wedge (v \leftarrow E)q$

= {definitions 3.37.3, 3.38.4} $wp_e(v := E)true \wedge wlp_e(v := E)q$.

4. S :: $S_1;S_2$

H1. $wp_e(S_1)q = wp_e(S_1)true \wedge wlp_e(S_1)q$.

H2. $wp_e(S_2)q = wp_e(S_2)true \wedge wlp_e(S_2)q$.

$wp_e(S_1;S_2)q$

= {definition 3.37.4} $wp_e(S_1)wp_e(S_2)q$

= {H1} $wp_e(S_1)true \wedge wlp_e(S_1)wp_e(S_2)q$

= {H2} $wp_e(S_1)true \wedge wlp_e(S_1)(wp_e(S_2)true \wedge wlp_e(S_2)q)$

= {theorem 3.42.2}

$wp_e(S_1)true \wedge wlp_e(S_1)wp_e(S_2)true \wedge wlp_e(S_1)wlp_e(S_2)$

= {H1} $wp_e(S_1)wp_e(S_2)true \wedge wlp_e(S_1)wlp_e(S_2)q$

= {definitions 3.37.4, 3.38.4} $wp_e(S_1;S_2)true \wedge wlp_e(S_1;S_2)q$.

5. S :: <u>if</u> $B_1 \to S_1 \;[]\; \ldots \;[]\; B_n \to S_n$ <u>fi</u>

H3. For all i: $1 \le i \le n$: $wp_e(S_i)q = wp_e(S_i)true \wedge wlp_e(S_i)q$.

$wp_e(S)q$

= {definition 3.37.5}

$[\vee i \mid 1 \le i \le n \mid B_i] \wedge [\wedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)q]$

= {H3}

$[\vee i \mid 1 \le i \le n \mid B_i] \wedge [\wedge i \mid 1 \le i \le n \mid B_i \Rightarrow (wp_e(S_i)true \wedge wlp(S_i)q)]$

= {prop. log.}

$[\vee i \mid 1 \le i \le n \mid B_i] \wedge [\wedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)true]$

$\wedge [\wedge i \mid 1 \le i \le n \mid B_i \Rightarrow wlp_e(S_i)q]$

= {definitions 3.37.5, 3.38.5} $wp_e(S)true \wedge wlp(S)q$.

6. S :: <u>do</u> $B_1 \to S_1 \;[]\; \ldots \;[]\; B_n \to S_n$ <u>od</u>

H4. For all i: $1 \le i \le n$: $wp_e(S_i)q = wp_e(S_i)true \wedge wlp_e(S_i)q$.

The proof that $(\mu F)(q) = (\mu F)(true) \wedge (\nu G)(q)$ is by simultaneous fixed point induction. Admissibility follows from lemma 3.44 below.

6.1. {base step}

$\bot_{T_e}(q)$

$= (\lambda q \in C_e \mid false)q$

$= false$

$= false \wedge true$

$= (\lambda q \in C_e \mid false)true \wedge (\lambda q \in C_e \mid true)q$

$= \bot_{T_e}(true) \wedge \tau_{T_e}(q)$.

6.2. {induction step}

H5. $f(q) = f(true) \wedge g(q)$.

$F(f)(g)$

= {definition 3.37.6}

$([\vee i \mid 1 \le i \le n \mid B_i] \vee q) \wedge [\wedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)f(q)]$

$= \{H4,H5\}$

$([\vee i \mid 1 \leq i \leq n \mid B_i] \vee q) \wedge [\wedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow$

$(wp_e(S_i)true \wedge wlp_e(S_i)(f(true) \wedge g(q)))]$

$= \{theorem\ 3.42.2\}$

$([\vee i \mid 1 \leq i \leq n \mid B_i] \vee q) \wedge [\wedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow$

$(wp_e(S_i)true \wedge wlp_e(S_i)(f(true)) \wedge wlp_e(S_i)(g(q)))]$

$= \{H4\}$

$([\vee i \mid 1 \leq i \leq n \mid B_i] \vee q) \wedge [\wedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow$

$(wp_e(S_i)(f(true)) \wedge wlp_e(S_i)(g(q)))]$

$= \{prop.\ log.\}$

$([\vee i \mid 1 \leq i \leq n \mid B_i] \vee q)$

$\wedge [\wedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow wp_e(S_i)(f(true))]$

$\wedge ([\vee i \mid 1 \leq i \leq n \mid B_i] \vee q)$

$\wedge [\wedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow wlpe(S_i)(g(q))]$

$= \{definitions\ 3.37.6,\ 3.38.6\}\ F(f)(true) \wedge G(g)(q)$.

□

The admissibility for simultaneous fixed point induction in proof 3.43.6 is an immediate consequence of the following lemma.

Lemma 3.44

Let $(C,\sqsubseteq)$ be a ccl.
Let $\langle f_i \rangle_{i=0}^{\infty}$ and $\langle h_i \rangle_{i=0}^{\infty}$ be two ascending chains in $(C,\sqsubseteq)$.
Let $\langle g_i \rangle_{i=0}^{\infty}$ be a descending chain in $(C,\sqsubseteq)$.
If $(\underline{A}\ i \mid i \geq 0 \mid f_i = h_i \sqcap g_i)$, then

$$\bigsqcup_{i=0}^{\infty} f_i = \bigsqcup_{i=0}^{\infty} h_i \sqcap \bigsqcap_{i=0}^{\infty} g_i \ .$$

□

Proof

1. $\{\sqsubseteq\}$

$$
\begin{aligned}
& f \text{ is ascending} \\
= \; & (\underline{A}\, i \mid i \geq 0 \mid (\underline{A}\, j \mid j \geq i \mid f_i \sqsubseteq f_j)) \\
= \; & \{(\underline{A}\, j \mid j \geq 0 \mid f_j = h_j \sqcap g_j)\} \\
& (\underline{A}\, i \mid i \geq 0 \mid (\underline{A}\, j \mid j \geq i \mid f_i \sqsubseteq h_j \sqcap g_j)) \\
= \; & \{\text{definition glb}\} \\
& (\underline{A}\, i \mid i \geq 0 \mid (\underline{A}\, j \mid j \geq i \mid f_i \sqsubseteq h_j \wedge f_i \sqsubseteq g_j)) \\
\Rightarrow \; & \{\text{definition lub, glb}\} \\
& (\underline{A}\, i \mid i \geq 0 \mid f_i \sqsubseteq \bigsqcup_{j=i}^{\infty} h_j \wedge f_i \sqsubseteq \bigsqcap_{j=i}^{\infty} g_j) \\
\Rightarrow \; & \{h \text{ ascending, } g \text{ descending}\} \\
& (\underline{A}\, i \mid i \geq 0 \mid f_i \sqsubseteq \bigsqcup_{j=0}^{\infty} h_j \wedge f_i \sqsubseteq \bigsqcap_{j=0}^{\infty} g_j) \\
\Rightarrow \; & \{\text{definition glb}\} \\
& (\underline{A}\, i \mid i \geq 0 \mid f_i \sqsubseteq \bigsqcup_{j=0}^{\infty} h_j \sqcap \bigsqcap_{j=0}^{\infty} g_j) \\
\Rightarrow \; & \{\text{definition lub}\} \\
& \bigsqcup_{i=0}^{\infty} f_i \sqsubseteq \bigsqcup_{j=0}^{\infty} h_j \sqcap \bigsqcap_{j=0}^{\infty} g_j \\
= \; & \{\text{renaming}\} \\
& \bigsqcup_{i=0}^{\infty} f_i \sqsubseteq \bigsqcup_{i=0}^{\infty} h_i \sqcap \bigsqcap_{i=0}^{\infty} g_i \; .
\end{aligned}
$$

2. $\{\sqsupseteq\}$

$$
\begin{aligned}
& \text{true} \\
= \; & \{\text{definition glb}\} \\
& (\underline{A}\, i \mid i \geq 0 \mid g_i \sqsupseteq \bigsqcap_{j=0}^{\infty} g_j) \\
\Rightarrow \; & \{\text{definition glb}\} \\
& (\underline{A}\, i \mid i \geq 0 \mid h_i \sqcap g_i \sqsupseteq h_i \sqcap \bigsqcap_{j=0}^{\infty} g_j) \\
= \; & \{(\underline{A}\, i \mid i \geq 0 \mid f_i = h_i \sqcap g_i)\} \\
& (\underline{A}\, i \mid i \geq 0 \mid f_i \sqsupseteq h_i \sqcap \bigsqcap_{j=0}^{\infty} g_j)
\end{aligned}
$$

{definition lub}

$\bigsqcup_{i=0}^{\infty} f_i \sqsupseteq \bigsqcup_{i=0}^{\infty} (h_i \sqcap \bigsqcap_{j=0}^{\infty} g_j)$

= {lemma 3.21}

$\bigsqcup_{i=0}^{\infty} f_i \sqsupseteq \bigsqcup_{i=0}^{\infty} h_i \sqcap \bigsqcap_{j=0}^{\infty} g_j$.

□

Theorem 3.45

Let $e \in \mathit{Env}$, $p,q \in C_e$; $S_1,\ldots,S_n \in L(\text{Stat } \langle e \rangle)$;
$B_1,\ldots,B_n \in L(\text{Expr } \langle e,\mathit{Prio},\text{bool} \rangle)$.
Let IF = **if** $B_1 \to S_1 \;[]\; \ldots \;[]\; B_n \to S_n$ **fi** .

1. If for all i: $1 \le i \le n$: $p \wedge B_i \sqsubseteq_e wp_e(S_i)q$, then

   $p \wedge [\bigvee i \mid 1 \le i \le n \mid B_i] \sqsubseteq_e wp_e(\text{IF})q$.

2. If for all i: $1 \le i \le n$: $p \wedge B_i \sqsubseteq_e wlp_e(S_i)q$, then

   $p \sqsubseteq_e wlp_e(\text{IF})q$.

□

Proof

We only prove 1. The proof of 2 is similar.

1. Assume

   A: for all i: $1 \le i \le n$: $p \wedge B_i \sqsubseteq_e wp_e(S_i)q$.

   $p \wedge [\bigvee i \mid 1 \le i \le n \mid B_i]$

   = {prop. log.}

   $[\bigvee i \mid 1 \le i \le n \mid B_i] \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow p \wedge B_i]$

   $\sqsubseteq_e$ {A}

   $[\bigvee i \mid 1 \le i \le n \mid B_i] \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp_e(S_i)q]$

   = {definition 3.37.5} $wp_e(\text{IF})q$.

□

Theorem 3.46

Let $e \in \mathit{Env}$; $p \in C_e$; $S_1,\ldots,S_n \in L(\text{Stat } \langle e \rangle)$;
$B_1,\ldots,B_n \in L(\text{Expr } \langle e,\mathit{Prio},\text{bool} \rangle)$.

Let $DO = \underline{do}\ B_1 \rightarrow S_1 \ [] \ \ldots \ [] \ B_n \rightarrow S_n\ \underline{od}$.

Let $BB = [\bigvee i \mid 1 \leq i \leq n \mid B_i]$.

1. If for all i: $1 \leq i \leq n$: $p \wedge B_i \sqsubseteq_e wp_e(S_i)p$, then

$$p \wedge wp_e(DO)true \sqsubseteq_e wp_e(DO)(p \wedge \neg BB) .$$

2. If for all i: $1 \leq i \leq n$: $p \wedge B_i \sqsubseteq_e wlp_e(S_i)p$, then

$$p \sqsubseteq_e wlp_e(DO)(p \wedge \neg BB) .$$

□

Proof

We first prove 2, and subsequently 1.

2. Assume

A2: for all i: $1 \leq i \leq n$: $p \wedge B_i \sqsubseteq_e wlp_e(S_i)p$.

The proof that $p \sqsubseteq_e (\nu G)(p \wedge \neg BB)$ is by greatest fixed point induction. Admissibility follows from definition 3.2.1.

2.1. {base step}

$$\begin{aligned} & p \\ \sqsubseteq_e\ & true \\ =\ & (\lambda q \in C_e \mid true)(p \wedge \neg BB) \\ =\ & \top_{T_e}(p \wedge \neg BB) . \end{aligned}$$

2.2. {induction step}

H: $p \sqsubseteq_e g(p \wedge \neg BB)$.

$$\begin{aligned} & G(g)(p \wedge \neg BB) \\ =\ & \{\text{definition 3.38.6}\} \\ & (BB \vee (p \wedge \neg BB)) \wedge [\bigwedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow \\ & \qquad wlp_e(S_i)(g(p \wedge \neg BB))] \\ =\ & \{\text{prop. log.}\} \\ & (BB \vee p) \wedge [\bigwedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow wlp_e(S_i)(g(p \wedge \neg BB))] \end{aligned}$$

$\sqsupseteq_e$ {H, corollary 3.40.2}

$(BB \vee p) \wedge [\wedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow wlp_e(S_i)p]$

$\sqsupseteq_e$ {A2}

$(BB \vee p) \wedge [\wedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow p \wedge B_i]$

$=$ {prop. log.} p .

1. Assume

A1: for all i: $1 \leq i \leq n$: $p \wedge B_i \sqsubseteq_e wp_e(S_i)p$.

By A1 and theorem 3.43: for all i: $1 \leq i \leq n$:

$$p \wedge B_i \sqsubseteq_e wp_e(S_i)true \wedge wlp_e(S_i)p ,$$

hence:

A1': for all i: $1 \leq i \leq n$: $p \wedge B_i \sqsubseteq_e wlp_e(S_i)p$.

$p \wedge wp_e(DO)true$

$\sqsubseteq_e$ {A1',A2}

$wlp_e(DO)(p \wedge \neg BB) \wedge wp_e(DO)true$

$=$ {theorem 3.43}

$wp_e(DO)(p \wedge \neg BB)$.

□

Theorem 3.47

Let $e \in \mathit{Env}$; $q \in C_e$; $\{p_\alpha \mid \alpha \geq 0\}$ a countable subset of C_e; $S_1,\ldots,S_n$, $B_1,\ldots,B_n$, IF, DO, BB as in theorems 3.45 and 3.46.

1. $wp_e(DO)q = (q \wedge \neg BB) \vee wp_e(IF)wp(DO)q$.

2. If for all α: $\alpha \geq 0$:

   a. for all i: $1 \leq i \leq n$: $p_\alpha \wedge B_i \sqsubseteq_e wp_e(S_i)[\vee \alpha' \mid 0 \leq \alpha' < \alpha \mid p_{\alpha'}]$

   b. $p_\alpha \wedge \neg BB \sqsubseteq q$

   then $[\vee \alpha \mid \alpha \geq 0 \mid p_\alpha] \sqsubseteq_e wp_e(DO)q$.

□

Proof

1. $wp_e(DO)q$

   = {definition 3.37.6, fixed point property}

   $(q \vee BB) \wedge [\wedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow wp_e(S_i)wp_e(DO)q]$

   = {definition BB, prop. log.}

   $(q \wedge \neg BB) \vee (BB \wedge [\wedge i \mid 1 \leq i \leq n \mid B_i \Rightarrow wp_e(S_i)wp_e(DO)q\ )]$

   = {definition 3.37.5}

   $(q \wedge \neg BB) \vee wp_e(IF)wp_e(DO)q$.

2. Let α: $\alpha \geq 0$. Consider

   $[\vee\alpha' \mid 0 \leq \alpha' < \alpha \mid p_{\alpha'}] \sqsubseteq_e wp_e(DO)q$

   $\Rightarrow$ {corollary 3.40.1}

   $wp_e(IF)[\vee\alpha' \mid 0 \leq \alpha' < \alpha \mid p_{\alpha'}] \sqsubseteq_e wp_e(IF)wp_e(DO)q$

   $\Rightarrow$ {a, theorem 3.45}

   $p_\alpha \wedge BB \sqsubseteq_e wp_e(IF)wp_e(DO)q$

   $\Rightarrow$ {b}

   $(p_\alpha \wedge \neg BB) \vee (p_\alpha \wedge BB) \sqsubseteq_e (q \wedge \neg BB) \vee wp_e(IF)wp_e(DO)q$

   = {prop. log., 1}

   $p_\alpha \sqsubseteq_e wp_e(DO)q$.

   We have shown that for all α: $\alpha \geq 0$:

   $[\vee\alpha' \mid 0 \leq \alpha' < \alpha \mid p_{\alpha'}] \sqsubseteq_e wp_e(DO)q \Rightarrow (p_\alpha \sqsubseteq_e wp_e(DO)q)$,

   hence by lemma 3.29:

   $[\vee\alpha \mid \alpha \geq 0 \mid p_\alpha] \sqsubseteq_e wp_e(DO)q$.

□

3.3. Logics for partial and total correctness

In this section we will extend the logic D with additional formulae and proof rules that enable us to prove partial and total correctness properties in the style of Hoare [Hoare 1, Hoare 2]. We will also prove consistency of these logics with respect to D. To this end we introduce notions of validity and soundness. It should be noted that our use of these terms differs from that in program correctness theory [de Bakker, Cook 2] or formal logic. In the latter fields these notions pertain to the relations with a model; in our case they pertain to relations with a model; in our case they pertain to relations with the logic D.

Partial and total correctness formulae are introduced by the following extensions to the attribute grammar for D-formulae.

Nonterminals

{Pform <*Env*>, Tform <*Env*>}.

Terminals

{ "{","}","[","]" }.

Grammar rules

Pform <e> ::= Form <e> ■

Pform <e> ::= Cont <e> | {Cond <e>} Stat <e> {Cond <e>} ■

Tform <e> ::= Form <e> ■

Tform <e> ::= Cont <e> | [Cond <e>] Stat <e> [Cond <e>] ■

Definition 3.48 {Pvalid0, Tvalid0}
On L(Pform <*Env*>) and L(Tform <*Env*>), respectively, the predicates Pvalid0 and Tvalid0 are defined as follows:

For all e $\in$ *Env*, c $\in L$(Cont <e>), p,q $\in L$(Cond <e>), S $\in S$(Stat <e>):

1.1. Pvalid0(c | p) = $\vdash_D$ c | p .

1.2. Pvalid0(c | {p}S{q}) = $\vdash_D$ c | p $\Rightarrow$ wlp_e(S)q .

2.1. $\mathrm{Tvalid0}(c \mid p) = \vdash_D c \mid p$.

2.2. $\mathrm{Tvalid0}(c \mid [p]S[q]) = \vdash_D c \mid p \Rightarrow \mathrm{wp}_e(S)q$.

□

Definition 3.49 {proof rule}

1. A partial correctness proof rule is a construct of the form

$$\frac{c_0 \mid f_0,\ldots,f_{n-1}}{c_1 \mid f_n}$$

where $n \geq 1$, and $c_0 \mid f_0,\ldots,c_0 \mid f_{n-1}$, $c_1 \mid f \in L(\mathrm{Pform}\ \langle Env \rangle)$.

2. A total correctness proof rule is a construct of the form

$$\frac{c_0 \mid f_0,\ldots,f_{n-1}}{c_1 \mid f_n}$$

where $n \geq 1$, and $c_0 \mid f_0,\ldots,c_0 \mid f_{n-1}$, $c_1 \mid f \in L(\mathrm{Tform}\ \langle Env \rangle)$.

□

Definition 3.50 {Psound0, Tsound0}

1. On partial correctness proof rules the predicate Psound0 is defined by:

$$\mathrm{Psound0}\left(\frac{c_0 \mid f_0,\ldots,f_{n-1}}{c_1 \mid f_n}\right)$$

$$= (\underline{A}\ i \mid 0 \leq i \leq n \mid \mathrm{Pvalid0}(c_0 \mid f_i)) \Rightarrow \mathrm{Pvalid0}(c_1 \mid f_n) .$$

2. On total correctness proof rules the predicate Tsound0 is defined by:

$$\mathrm{Tsound0}\left(\frac{c_0 \mid f_0,\ldots,f_{n-1}}{c_1 \mid f_n}\right)$$

$$= (\underline{A}\ i \mid 0 \leq i \leq n \mid \mathrm{Tvalid0}(c_0 \mid f_i)) \Rightarrow \mathrm{Tvalid0}(c_1 \mid f_n)) .$$

□

Definition 3.51 $\{PC_0, PA_1,\ldots,PA_3, PR_1,\ldots,PR_4\}$

The partial correctness logic PC_0 is defined by

- $Ax_{PC_0} = Ax_D \cup \{PA_1,PA_2,PA_3\}$
- $Pr_{PC_0} = Pr_D \cup \{PR_1,PR_2,PR_3,PR_4\}$

where $PA_1,\ldots,PA_3$, $PR_1,\ldots,PR_4$ are given below.

For all $e \in \mathit{Env}$,
$c \in L(\text{Cont } \langle e\rangle)$,
$p,q,q_1,q_2,q_3,q_4 \in L(\text{Cond } \langle e\rangle)$,
$B_1,\ldots,B_n \in L(\text{Expr } \langle e,\mathit{Prio},\text{bool}\rangle)$,
$S,S_1,\ldots,S_n \in L(\text{Stat } \langle e\rangle)$,
v, E such that $v := E \in L(\text{Stat } \langle e\rangle)$:

PA_1. $c \mid \{\text{true}\}\ \text{abort}\ \{q\}$

PA_2. $c \mid \{q\}\ \text{skip}\ \{q\}$

PA_3. $c \mid \{(v \leftarrow E)q\}\ v := E\ \{q\}$

$$PR_1.\ \frac{c \mid q_1 \Rightarrow q_2,\ \{q_2\}\ S\ \{q_3\},\ q_3 \Rightarrow q_4}{c \mid \{q_1\}\ S\ \{q_4\}}$$

$$PR_2.\ \frac{c \mid \{q_1\}\ S_1\ \{q_2\},\ \{q_2\}\ S_2\ \{q_3\}}{c \mid \{q_1\}\ S_1; S_2\ \{q_3\}}$$

$$PR_3.\ \frac{c \mid \{p \wedge B_1\}\ S_1\ \{q\}\ ,\ldots,\ \{p \wedge B_n\}\ S_n\ \{q\}}{c \mid \{p\}\ \underline{\text{if}}\ B_1 \to S_1\ []\ \ldots\ []\ B_n \to S_n\ \underline{\text{fi}}}$$

$$PR_4.\ \frac{c \mid \{p \wedge B_1\}\ S_1\ \{p\}\ ,\ldots,\ \{p \wedge B_n\}\ S_n\ \{p\}}{c \mid \{p\}\ \underline{\text{do}}\ B_1 \to S_1\ []\ \ldots\ []\ B_n \to S_n\ \underline{\text{od}}\ \{p \wedge \neg[\bigvee i \mid 1\le i\le n \mid B_i]\}}$$

□

Theorem 3.52

1. For all $ax \in Ax_{PC_0}$: Pvalid0(ax) .
2. For all $pr \in Pr_{PC_0}$: Psound0(pr) .

□

Proof

We only have to consider $PA_1,\ldots,PA_3$, $PR_1,\ldots,PR_4$.

1. Pvalid0(PA_1), Pvalid0(PA_2) and Pvalid0(PA_3) follow immediately from definition 3.48.1.2 and definitions 3.38.1, 3.38.2 and 3.38.3, respectively.

2.1. case PR_1

Assume A1. Pvalid0(c | $q_1 \Rightarrow q_2$) .

A2. Pvalid0(c | $\{q_2\}$ S $\{q_3\}$) .

A3. Pvalid0(c | $q_3 \Rightarrow q_4$) .

$wlp_e(S)q_4$

$\exists_e$ {A3, corollary 3.40.2} $wlp_e(S)q_3$

$\exists_e$ {A2} q_2

$\exists_e$ {A1} q_1

hence Pvalid0(c | $\{q_1\}$ S $\{q_4\}$) .

2.2. case PR_2

Assume A1. Pvalid0(c | $\{q_1\}$ S_1 $\{q_2\}$) .

A2. Pvalid0(c | $\{q_2\}$ S_2 $\{q_3\}$) .

$wlp_e(S_1;S_2)q_3$

= {definition 3.39.4} $wlp_e(S_1)wlp(S_2)q_3$

$\exists_e$ {A2, corollary 3.40.2} $wlp_e(S_1)q_2$

$\exists_e$ {A1} q_1

hence Pvalid0(c | $\{q_1\}$ $S_1;S_2$ $\{q_3\}$) .

2.3. case PR_3

Assume for all i: $1 \leq i \leq n$: Pvalid0(c | $\{p \wedge B_i\}$ S_i $\{q\}$).

By theorem 3.45.2:

Pvalid0(c | $\{p\}$ <u>if</u> $B_1 \rightarrow S_1$ [] ... [] $B_n \rightarrow S_n$ <u>fi</u> $\{q\}$) .

2.4. case PR_4

Assume for all i: $1 \le i \le n$: Pvalid0(c | {p ∧ B_i} S_i {p}).
By theorem 3.46.2:

Pvalid0(c | {p} do $B_1 \to S_1$ [] ... [] $B_n \to S_n$ od {p ∧ ¬[∨i | $1 \le i \le n$ | B_i]}

□

Definition 3.53 {TC_0, $TA_1,\ldots,TA_3$, $TR_1,\ldots,TR_4$}

The total correctness logic TC_0 is defined by

- $Ax_{TC_0} = Ax_D \cup \{TA_1, TA_2, TA_3\}$
- $Pr_{TC_0} = Pr_D \cup \{TR_1, TR_2, TR_3, TR_4\}$

where $TA_1,\ldots,TA_3$, $TR_1,\ldots,TR_4$ are given below.

For all e ∈ *Env*,
c ∈ L(Cont <e>),
p,q,q_1,q_2,q_3,q_4 ∈ L(Cond <e>),
$B_1,\ldots,B_n$ ∈ L(Expr <e,*Prio*,bool>),
$S,S_1,\ldots,S_n$ ∈ L(Stat <e>),
v, E such that v := E ∈ L(Stat <e>),
a ∈ L(Expr <e,*Prio*,int>), A ∈ *Name* such that ⊢ new(A,e) :

TA_1. c | [false] abort [q]

TA_2. c | [q] skip [q]

TA_3. c | [(v ← E)q] v := E [q]

$$TR_1.\ \frac{c \mid q_1 \Rightarrow q_2,\ [q_2]\ S\ [q_3],\ q_3 \Rightarrow q_4}{c \mid [q_1]\ S\ [q_4]}$$

$$TR_2.\ \frac{c \mid [q_1]\ S_1\ [q_2],\ [q_2]\ S_2\ [q_3]}{c \mid [q_1]\ S_1;S_2\ [q_3]}$$

$$TR_3.\ \frac{c \mid [p \wedge B_1]\ S_1\ [q]\ ,\ldots,\ [p \wedge B_n]\ S_n\ [q]}{c \mid [p \wedge [\vee i \mid 1 \le i \le n \mid B_i]]\ \underline{\text{if}}\ B_1 \to S_1\ []\ \ldots\ []\ B_n \to S_n\ \underline{\text{fi}}\ [q]}$$

$$TR_4.\ c \triangleright A\colon \mathrm{int} \mid [p \wedge B_1 \wedge 0{\le}a \wedge a{=}A]\ S_1\ [p \wedge 0{\le}a \wedge a{<}A]$$
$$\cdots$$
$$[p \wedge B_n \wedge 0{\le}a \wedge a{=}A]\ S_n\ [p \wedge 0{\le}a \wedge a{<}A]$$
$$\overline{c \mid [p \wedge 0{\le}a]\ \underline{\mathrm{do}}\ B_1 \to S_1\ []\ldots[]\ B_n \to S_n\ \underline{\mathrm{od}}\ [p \wedge \neg[\bigvee i \mid 1{\le}i{\le}n \mid B_i]]}$$

□

Theorem 3.54

1. For all $ax \in Ax_{TC_0}$: Tvalid0(ax) .
2. For all $pr \in Pr_{TC_0}$: Tsound0(pr) .

□

Proof

We only prove soundness of TR_4; the other proofs are very similar to those of theorem 3.52.

Let $e' = \mathrm{Ext}(e,[A,\mathrm{int}]_D)$.

for all i: $1 \le i \le n$:

Tvalid0($c \triangleright A\colon \mathrm{int} \mid [p \wedge B_i \wedge 0{\le}a \wedge a{=}A]\ S_i\ [p \wedge 0{\le}a \wedge a{<}A]$)

= {definition 3.48.2}

for all i: $1 \le i \le n$:

$\vdash_D c \triangleright A\colon \mathrm{int} \mid (p \wedge B_i \wedge 0{\le}a \wedge a{=}A) \Rightarrow wp_{e'}(S_i)(p \wedge 0{\le}a \wedge a{<}A)$

= {axiom of D}

$\vdash_D c \mid (\underline{A}\ A\colon \mathrm{int} \parallel (p \wedge B_i \wedge 0{\le}a \wedge a{=}A) \Rightarrow wp_{e'}(S_i)(p \wedge 0{\le}a \wedge a{<}A))$

⇒ {*see note below}

for all i: $1 \le i \le n$:

for all α: $0 \le \alpha$:

$\vdash_D c \mid (p \wedge B_i \wedge a{=}\tilde{\alpha}) \Rightarrow wp_e(S_i)(p \wedge 0{\le}a \wedge a{<}\tilde{\alpha})$

⇒ {definition 3.33, theorem 3.47}

$[\bigvee \alpha \mid 0{\le}\alpha \mid p \wedge B_i \wedge a{=}\tilde{\alpha}] \sqsubseteq_e wp_e(DO)(p \wedge \neg BB)$

= $p \wedge 0{\le}a \sqsubseteq_e wp_e(DO)(p \wedge \neg BB)$

= {definition 3.48.2}

Tvalid0($c \mid [p \wedge 0{\le}a]$ DO $[p \wedge \neg BB]$) .

□

Note

In the step marked with a * we have applied the rule for A-elimination. The notation $\tilde{\alpha}$ stands for an element of L(Con <int>) representing the value α. As in the resulting conditions the variable A no longer occurs, wp_e, may be replaced by wp_e. Replacements of the latter kind will be discussed more extensively in chapter 4.

□

CHAPTER 4
BLOCKS AND PROCEDURES

4.0. Introduction

In this chapter we consider the design and formal definition of some essential components of the source language, viz. blocks and procedures. We do so for various reasons:

- Blocks and procedures are nontrivial extensions to the kernel language of chapter 3. They pose many interesting problems with regard to the construction of correct compilers, which is the ultimate goal of our studies.
- In the literature on formal definitions in many cases the treatment of blocks and procedures is either incomplete or very complex. The incompleteness usually results from considering either the syntactic or the semantic aspects of the constructs, whereas their interaction is often essential (e.g. in proof rules for blocks, which critically depend on the scopes of variables). The complexity usually results from considering an existing procedure concept as it occurs in, say, ALGOL 60 or Pascal, as holy, and trying to formalize all its aspects as faithfully as possible, without questioning the quality of that concept. In contrast, we want to define completely both syntax and semantics, but we will try to design language constructs in such a way that their definition is relatively simple.
- It is interesting to investigate to what extent the condition transformer method, which was designed for statements, can be applied to more complex language constructs like recursive procedures.

We will try to separate the various aspects as much as possible, and to study their effect on syntax, semantics and proof rules. This separation is reflected in the structure of the chapter. In section 4.1 we will discuss blocks, mainly to investigate the effects of the

introduction of local names. Section 4.2 deals with parameter mechanisms. The discussion is based on a language construct called abstraction, which resembles the ALGOL 68 routinetext, and which can be used to study the effects of parametrization. Section 4.3 concentrates on recursion, which can be handled rather easily by means of the lattice theory of section 3.1. Finally, in section 4.4 the various aspects are merged, resulting in a treatment of parametrized recursive procedures. Each section follows the same pattern of discussing first syntax, then semantics, and finally proof rules.

4.1. Blocks

4.1.0. Introduction

In section 4.1 we discuss the block, a construct which provides the means for local extension of the environment of statements. Although the block is of some interest in its own right, our prime motivation to discuss it is the desire to separate the aspect of the introduction of local nomenclature from other aspects of the procedure concept.

From section 2.3.2 we recall the grammar rule for blocks:

2. Block $\langle e_0 \rangle$ ::= |[var Decs $\langle d \rangle$ | Stat $\langle e_1 \rangle$]| ■
   (A n: *Name* | $\#_D$ (n,d) ≤ 1)
   e_1 = Ext(e_0,d)

Throughout section 4.1 we shall restrict ourselves to blocks containing a single variable declaration, i.e. blocks of the form |[var x: t | S]|. Generalization to other blocks is straightforward.

In section 4.1.1 we consider the case that redeclaration is not allowed. As a preparation for the treatment of redeclaration in section 4.1.3 and for other sections, section 4.1.2 is devoted to substitution in programming language constructs. In section 4.1.4 proof rules for blocks are presented and their soundness is proven.

4.1.1. Blocks without redeclaration

In this section we study the semantics of a block |[var x: t | S]| $\in L$(Block $\langle e_0 \rangle$) under the additional assumption that $\vdash$ new(x,e_0), i.e. that x has not been declared in surrounding blocks. What we need is a definition of wp_{e_0}(|[var x: t | S]|) in terms of wp_{e_1}(S). Since it is our intention that |[var x: t | S]| and S have the same effect as far as the variables of e_0 are concerned, a definition of the following form readily suggests itself:

$$wp_{e_0}(|[\ \underline{\text{var}}\ x\colon t \mid S\ ]|) = (\lambda q \in C_{e_0} \mid wp_{e_1}(S)q)\ .$$

However, this definition is not always correct. Because e_1 is an extension of e_0, C_{e_0} is a proper subset of C_{e_1}. Therefore $wp_{e_1}(S)$ may be applied to q, but the yield is not necessarily an element of C_{e_0}, since it might still contain x. A little reflection reveals that this situation will only occur when x has not been initialized by S (where for the moment we assume that we know what is meant by initialization). There are at least two solutions to the problem:

1. Require that S establishes the desired post-condition regardless of the initial value of x. This boils down to universal quantification over x, as a result of which the pre-condition becomes an element of C_{e_0}:

$$wp_{e_0}(|[\ \underline{var}\ x\colon t \mid S\ ]|) = (\lambda q \in C_{e_0} \mid (\underline{A}\ x\colon t \parallel wp_{e_1}(S)q)) .$$

2. Impose the additional context condition that S initializes x.

Solution 1 does not fit well into the framework developed in chapter 3 because the resulting condition transformer is not upward continuous. In the terminology of [Back 1, Dijkstra 2] a block containing a variable of an unbounded type would be a construct of "unbounded nondeterminacy"; e.g. the block |[var y: nat | x := y+1]| would be an implementation of Dijkstra's "set x to any positive integer" [Dijkstra 2]. Because of this complication we will not adopt solution 1.

Solution 2 is in accordance with "disciplined" programming. Systematically constructed programs will never contain uninitialized variables and therefore we may as well exclude them syntactically. For the duration of this chapter context conditions of this kind will be expressed by means of some auxiliary functions defined below. Eventually they will be incorporated into the attribute grammar for the source language.

Definition 4.1 {USE}
The function USE

$$L(\text{Stat } \langle Env \rangle) \cup L(\text{Expr } \langle Env, Prio, Type \rangle) \cup L(\text{Cond } \langle Env \rangle) \to P(Name)$$

is defined recursively by

1. $USE(\text{abort}) = \emptyset$

2. $USE(\text{skip}) = \emptyset$

3. $USE(v := E) = USE(E)$

4. $USE(S_1;S_2) = USE(S_1) \cup USE(S_2)$

5. $USE(\underline{\text{if}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{fi}}) = \bigcup_{i=1}^{n} USE(B_i) \cup \bigcup_{i=1}^{n} USE(S_i)$

6. $USE(\underline{\text{do}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{od}}) = \bigcup_{i=1}^{n} USE(B_i) \cup \bigcup_{i=1}^{n} USE(S_i)$

7. $USE(|[\ \underline{\text{var}}\ x\text{: } t \mid S\ ]|) = USE(S) \setminus \{x\}$

8. $USE(E_1\ \text{op}\ E_2) = USE(E_1) \cup USE(E_2)$
   for op $\in L(\text{Dop } \langle Prio, Type, Type, Type\rangle)$

9. $USE(\text{op } E) = USE(E)$ for op $\in L(\text{Mop } \langle Type, Type\rangle)$

10. $USE((E)) = USE(E)$

11. $USE(v) = \{v\}$ for v $\in L(\text{Var } \langle Env, Name, Type\rangle)$

12. $USE(c) = \emptyset$ for c $\in L(\text{Con } \langle Type\rangle)$

13. $USE((\underline{A}\ x\text{: } t \mid E_1 \mid E_2)) = USE(E_1) \cup USE(E_2) \setminus \{x\}$

14. $USE((\underline{E}\ x\text{: } t \mid E_1 \mid E_2)) = USE(E_1) \cup USE(E_2) \setminus \{x\}$

15. $USE([\bigvee i \mid 0 \le i \mid q_i]) = \bigcup_{0 \le i} USE(q_i)$

16. $USE([\bigwedge i \mid 0 \le i \mid q_i]) = \bigcup_{0 \le i} USE(q_i)$

□

Definition 4.2 {ASSN}

The function ASSN: $L(\text{Stat } \langle Env\rangle) \to P(\underline{Name})$ is defined recursively by:

1. $ASSN(\text{abort}) = \emptyset$

2. $ASSN(\text{skip}) = \emptyset$

3. $ASSN(v := E) = \{v\}$

4. $ASSN(S_1;S_2) = ASSN(S_1) \cup ASSN(S_2)$

5. $ASSN(\underline{\text{if}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{fi}}) = \bigcup_{i=1}^{n} ASSN(S_i)$

6. $ASSN(\underline{\text{do}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{od}}) = \bigcup_{i=1}^{n} ASSN(S_i)$

7. ASSN(|[var x: t | S]|) = ASSN(S) \ {x}

□

Definition 4.3 {INIT}

The function INIT: $\mathcal{L}$(Stat <*Env*>) → $\mathcal{P}$(*Name*) is defined recursively by:

1. INIT(abort) = ∅
2. INIT(skip) = ∅
3. INIT(v := E) = {v} \ USE(E)
4. $INIT(S_1;S_2) = INIT(S_1) \cup (INIT(S_2) \setminus USE(S_1))$
5. $INIT(\underline{if}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{fi}) = \bigcap_{i=1}^{n} INIT(S_i) \setminus \bigcup_{i=1}^{n} USE(B_i)$
6. $INIT(\underline{do}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{od}) = \emptyset$
7. INIT(|[var x: t | S]|) = INIT(S) \ {x}

□

Informally, for a construct c the set USE(c) may be interpreted as the set of variables occurring in an expression in c, ASSN(c) as the set of variables occurring in the left-hand side of an assignment in c, and INIT(c) as the set of variables initialized by c, i.e. assigned to by every possible execution of c and not used in any expression before they have been assigned to. The set USE(c) ∪ ASSN(c) is the set of "free" variables of c.

Example

Let S be the statement

x := 0; if x ≥ y → y := 0; z := 0 [] true → w := w; z := x fi

USE(S) = {x,y,w}
ASSN(S) = {x,y,z,w}
INIT(S) = {x,z} .

□

Lemma 4.4

For all e $\in$ *Env*, S $\in$ L(Stat <e>):

1. INIT(S) $\subseteq$ ASSN(S) .
2. ASSN(S) $\subseteq$ {n $\in$ *Name* | ($\underline{E}$ t $\in$ *Type* | $\vdash$ (n,t) $\underline{in}_E$ e)} .
3. USE(S) $\subseteq$ {n $\in$ *Name* | ($\underline{E}$ t $\in$ *Type* | $\vdash$ (n,t) $\underline{in}_E$ e)} .

□

Proof

By induction on the composition of statements. Details omitted.

□

Now consider again the proposed definitions

$$wp_{e_0}(|[\ \underline{var}\ x\colon t \mid S\ ]|) = (\lambda q \in C_{e_0} \mid\ wp_{e_1}(S)q)$$

$$wlp_{e_0}(|[\ \underline{var}\ x\colon t \mid S\ ]|) = (\lambda q \in C_{e_0} \mid wlp_{e_1}(S)q) .$$

By induction on the composition of statements and the nesting depth of blocks it can easily be seen that these definitions are well-formed (i.e. $wp_{e_1}(S)q \in C_{e_0}$ and $wlp_{e_1}(S)q \in C_{e_0}$) if we impose the additional context condition:

$$\boxed{x \in INIT(S) \vee x \notin USE(S)}$$

A similar condition is formulated in [de Bakker].
For future reference we collect some properties of USE, ASSN, and INIT in the following lemma.

Lemma 4.5

For all e $\in$ *Env*, S $\in$ L(Stat <e>), p,q $\in$ L(Cond <e>):

1. $USE(wp_e(S)q) \subseteq USE(S) \cup USE(q)$.
2. $USE(wlp_e(S)q) \subseteq USE(S) \cup USE(q)$.
3. $USE(wp_e(S)q) \cap INIT(S) = \emptyset$.
4. $USE(wlp_e(S)q) \cap INIT(S) = \emptyset$.
5. If $ASSN(S) \cap USE(p) = \emptyset$, then $wp_e(S)(p \wedge q) = p \wedge wp_e(S)q$.
6. If $ASSN(S) \cap USE(p) = \emptyset$, then $wlp_e(S)(p \wedge q) = p \wedge wlp_e(S)q$.

□

Proof

By induction on the composition of statements. Details omitted.

□

4.1.2. Substitution in statements

In the sequel we will sometimes define the meaning of a construct in terms of the meaning of a second construct derived from the first one by a process involving systematic replacements of variables. Such an approach requires a precise definition of systematic replacement. Here we present such a definition, much resembling those in [Curry, de Bakker].

Definition 4.6 {(x ← y), substitution in statements}

For all x,y ∈ L(Var <*Env*,*Name*,*Type*>) the substitution operator (x ← y) on L(Stat <*Env*>) ∪ L(expr <*Env*,*Prio*,*Type*>) is defined recursively by:

1. (x ← y) abort = abort
2. (x ← y) skip = skip
3. (x ← y)(v := E) = (x ← y)v := (x ← y)E
4. (x ← y)(S_1;S_2) = (x ← y)S_1;(x ← y)S_2
5. (x ← y) **if** B_1 → S_1 [] ... [] B_n → S_n **fi** =

   if (x ← y)B_1 → (x ← y)S_1 [] ... [] (x ← y)B_n → (x ← y)S_n **fi**
6. (x ← y) **do** B_1 → S_1 [] ... [] B_n → S_n **od** =

   do (x ← y)B_1 → (x ← y)S_1 [] ... [] (x ← y)B_n → (x ← y)S_n **od**
7.
   1. if x = w, then

      (x ← y) |[**var** w: t | S]| = |[**var** w: t | S]|
   2. if x ≠ w and y ≠ w, then

      (x ← y) |[**var** w: t | S]| = |[**var** w: t | (x ← y)S]|
   3. if x ≠ w and y = w, then

      (x ← y) |[**var** w: t | S]| = |[**var** z: t | (x ← y)(w ← z)S]|

      where z is the first element (in some suitable ordering) of

L(Var <*Env*,*Prio*,*Type*>) such that

$z \notin \{x,y\} \cup USE(S) \cup ASSN(S)$.

8. $(x \leftarrow y)(E_1 \text{ op } E_2) = (x \leftarrow y)E_1 \text{ op } (x \leftarrow y)E_2$

9. $(x \leftarrow y)(\text{op } E) = \text{op } (x \leftarrow y)E$

10.
   1. $(x \leftarrow y)z = y$ if $x = z$
   2. $(x \leftarrow y)z = z$ if $x \neq z$

   $\Big\}$ $z \in L$(Var <*Env*,*Prio*,*Type*>)

11. $(x \leftarrow y)z = z$ $\quad z \in L$(Con <*Type*>)

□

We recall from section 3.2.4 that we assume that for conditions substitution has been defined in the customary way. Without proof we state the following obvious property.

Lemma 4.7

For all $x,y \in L$(Var <*Env*,*Name*,*Type*>), $s \in L$(Stat <*Env*>), $c \in L$(Expr <*Env*,*Prio*,*Type*>) $\cup$ L(Cond <*Type*>):

- if $x \notin USE(s) \cup ASSN(s)$, then $(x \leftarrow y)s = s$.
- if $x \notin USE(c)$, then $(x \leftarrow y)c = c$.

□

The following lemma relates the condition transformers of a statement S and a statement S' obtained from S by systematic replacement of a variable x by a variable x'.

Lemma 4.8

Let $e \in$ *Env*, $S \in L$(Stat <e>).
Let $x,x' \in$ *Name* such that $\vdash \neg$ new(x,e), $\vdash$ new(x',e).
Let $e' \in$ *Env* such that $\vdash$ rep(e,e',x,x').
Let $S' = (x \leftarrow x')S$.

1. $wp_e(S) = (x' \leftarrow x) \circ wp_{e'}(S') \circ (x \leftarrow x')$.
2. $wlp_e(S) = (x' \leftarrow x) \circ wlp_{e'}(S') \circ (x \leftarrow x')$.

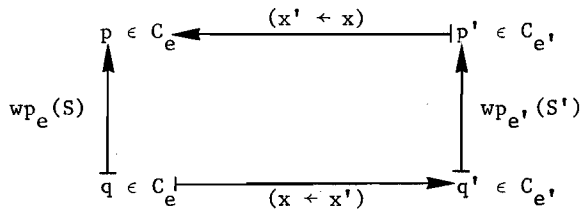
□

Proof

By induction on the composition of statements. Details omitted.

□

The following diagram may help in understanding lemma 4.8:



Lemma 4.8 will now be used to prove the following lemma, which states that the local variable of a block may be replaced by a different variable without affecting the meaning of the block, provided that the replacement does not lead to violation of the condition on redeclaration. This replacement is comparable to the α-conversion of lambda calculus.

Lemma 4.9

Let e ∈ *Env*, y ∈ *Name* such that ⊢ new(y,e).

Let ⟦ var x: t | S ⟧ ∈ *L*(Stat <e>).

1. wp_e (⟦ var x: t | S ⟧) = wp_e (⟦ var y: t | (x ← y)S ⟧) .
2. wlp_e(⟦ var x: t | S ⟧) = wlp_e(⟦ var y: t | (x ← y)S ⟧) .

□

Proof

We only prove 1.

Let e_1 = Ext(e,$[x,t]_D$), e_2 = Ext(e,$[y,t]_D$). Let q ∈ C_e.

wp_e(⟦ var x: t | S ⟧)q

= {proposed definition in section 4.1.1}

wp_{e_1}(S)q

= {lemma 4.8}

(y ← x)wp_{e_2}((x ← y)S)(x ← y)q

$= \{x \notin USE(q),$ lemma 4.7$\}$

$(y \leftarrow x)wp_{e_2}((x \leftarrow y)S)q$

$= \{y \in INIT((x \leftarrow y)S) \vee y \notin USE((x \leftarrow y)S),\ y \notin USE(q),$ lemmas 4.5, 4,7$\}$

$wp_{e_2}((x \leftarrow y)S)q$

$= \{$proposed definition in section 4.1.1$\}$

$wp_e(|[\ \underline{var}\ y: t \mid (x \leftarrow y)S\ ]|)q$.

□

4.1.3. Blocks with the possibility of redeclaration

Finally we define the semantics of a block $|[\ \underline{var}\ x: t \mid S\ ]|$ $\in L(\text{Block } \langle e_0 \rangle)$ without the restriction $\vdash new(x,e_0)$. In this case the proposed definitions of section 4.1.1 are not directly applicable as they could result in name clashes between the local x of a block and a nonlocal x. Considering the purport of lemma 4.9, viz. that within certain limits the meaning of a block is not affected by systematic replacement of the local variable, we are led to the following solution:

Definition 4.10 {wp and wlp for blocks}
For all $e_0 \in \mathit{Env}$, $|[\ \underline{var}\ x: t \mid S\ ]| \in L(\text{Block } \langle e_0 \rangle)$ such that $x \in INIT(S) \vee x \notin USE(S)$:

1. $wp_{e_0}(|[\ \underline{var}\ x: t \mid S\ ]|) = (\lambda q \in C_{e_0} \mid wp_{e_1}((x \leftarrow y)S)q)$,
2. $wlp_{e_0}(|[\ \underline{var}\ x: t \mid S\ ]|) = (\lambda q \in C_{e_0} \mid wlp_{e_1}((x \leftarrow y)S)q)$,

where, in both cases,

$y \in \mathit{Name}$ such that $\vdash new(y,e_0)$,

$e_1 = Ext(e_0,[y,t]_D)$.

□

Note
It can easily be shown that the theorems of sections 3.2.4 and 3.3 also hold when the kernel language is extended with blocks as defined in this section. We will not give these proofs, but we will freely use the theorems where appropriate.

□

4.1.4. Proof rules

In this section we present some proof rules for blocks of the form ⟦ var x: t | S ⟧, and we prove soundness of these rules. The first two pairs - i.e. (TR5,PR5) and (TR6,PR6) - are intended for the case that x has also been declared in some surrounding block. Rule PR5 is essentially the block rule proposed by Hoare in [Hoare 2]. It makes use of systematic replacement in S of the variable x by a fresh variable x'. Soundness of PR5 follows trivially from definition 4.10.2. The obvious disadvantage is that its use requires substitutions in the program text. Rules TR6 and PR6 do not have this disadvantage; they are based on renaming of the nonlocal x. PR6 is essentially the block rule proposed by Lauer [Lauer]; it is also discussed in [Cook 2]. In the absence of redeclaration rules TR5 and PR5 can be simplified to TR7 and PR7, respectively.

Definition 4.11 {TR5, PR5, TR6, PR6, TR7, PR7}
For all $e \in \mathit{Env}$,
$⟦ \text{var } x\text{: } t \mid S ⟧ \in L(\text{Block } \langle e \rangle)$,
$x' \in \mathit{Name}$ such that $\vdash \text{new}(x',e)$,
$e' \in \mathit{Env}$ such that $\vdash \text{rep}(e,e',x,x')$,
$p,q \in L(\text{Cond } \langle e \rangle)$,
$c \in L(\text{Cont } \langle e \rangle)$, $c' \in L(\text{Cont } \langle e' \rangle)$,
$p' = (x \leftarrow x')p$, $q' = (x \leftarrow x')$, $S' = (x \leftarrow x')S$,

the proof rules TR5, PR5, TR6, PR6, TR7, PR7 are defined by:

$$\text{TR5.}\ \frac{c \triangleright x'\text{: } t \mid [p]\ S'\ [q]}{c \mid [p]\ ⟦ \text{var } x\text{: } t \mid S ⟧\ [q]}$$

$$\text{PR5}\ \frac{c \triangleright x'\text{: } t \mid \{p\}\ S'\ \{q\}}{c \mid \{p\}\ ⟦ \text{var } x\text{: } t \mid S ⟧\ \{q\}}$$

$$\text{TR6.}\ \frac{c' \triangleright x\text{: } t \mid [p']\ S\ [q']}{c \mid [p]\ ⟦ \text{var } x\text{: } t \mid S ⟧\ [q]}$$

$$\text{PR6}\ \frac{c' \triangleright x\text{: } t \mid \{p'\}\ S\ \{q'\}}{c \mid \{p\}\ ⟦ \text{var } x\text{: } t \mid S ⟧\ \{q\}}$$

$$\text{TR7}\quad \frac{c \triangleright x\colon t \mid [p]\ S\ [q]}{c \mid [p]\ [\![\ \underline{\text{var}}\ x\colon t \mid S\ ]\!]\ [q]}\ ,\quad \text{provided} \vdash \text{new}(x,e)$$

$$\text{PR7}\quad \frac{c \triangleright x\colon t \mid \{p\}\ S\ \{q\}}{c \mid \{p\}\ [\![\ \underline{\text{var}}\ x\colon t \mid S\ ]\!]\ \{q\}}\ ,\quad \text{provided} \vdash \text{new}(x,e)$$

□

Theorem 4.12

1. Tsound0 (TR5)
2. Psound0 (PR5)
3. Tsound0 (TR6)
4. Psound0 (PR6)
5. Tsound0 (TR7)
6. Psound0 (PR7) .

□

Proof

We only prove 1 and 3. The proofs of 2 and 4 are similar.
The proofs of 5 and 6 are special cases of 1 and 3, respectively.

1. Let $e_1 = \text{Ext}(e,[x',t]_D)$.

   Tvalid0($c \triangleright x'\colon t \mid [p]\ S'\ [q]$)

   = {definition 3.48.2.2}

   $\vdash_D c \triangleright x'\colon t \mid p \Rightarrow wp_{e_1}(S')q$

   = {definition 4.10.1}

   $\vdash_D c \triangleright x'\colon t \mid p \Rightarrow wp_e([\![\ \underline{\text{var}}\ x\colon t \mid S\ ]\!])q$

   = {$x' \notin \text{USE}(p \Rightarrow wp_e([\![\ \underline{\text{var}}\ x\colon t \mid S\ ]\!])q)$}

   $\vdash_D c \mid p \Rightarrow wp_e([\![\ \underline{\text{var}}\ x\colon t \mid S\ ]\!])q$

   = {definition 3.48.2.2}

   Tvalid0($c \mid [p]\ [\![\ \text{var}\ x\colon t \mid S\ ]\!]\ [q]$) .

3. Tvalid0($c' \triangleright x\colon t \mid [p']\ S\ [q']$)

   ⇒ {theorem 4.12.1 with x and x' interchanged}

   Tvalid0($c' \mid [p']\ [\![\ \underline{\text{var}}\ x\colon t \mid S\ ]\!]\ [q']$

= {definition 3.48.2.2}

$\vdash_D c' \mid p' \Rightarrow wp_{e'}(|[\ \underline{var}\ x\colon t \mid S\ ]|)q'$

= {definition 4.6.7.1}

$\vdash_D c' \mid p' \Rightarrow wp_{e'}((x \leftarrow x')\ |[\ \underline{var}\ x\colon t \mid S\ ]|)q'$

= {$\vdash$ new(x,e')}

$\vdash_D c \mid (x' \leftarrow x)p' \Rightarrow (x' \leftarrow x)wp_{e'}((x \leftarrow x')\ |[\ \underline{var}\ x\colon t \mid S\ ]|)q'$

= {definition p',q'; lemma 4.8.1}

$\vdash_D c \mid p \Rightarrow wp_e(|[\ \underline{var}\ x\colon t \mid S\ ]|)q$

= {definition 3.48.2.2}

Tvalid0(c | [p] |[var x: t | S]| [q]) .

□

4.2. Abstraction and application

4.2.0. Introduction

Another aspect of the procedure concept we want to study in isolation is parameterization. To this end we introduce in this section a new language construct, called abstraction, which somewhat resembles the lambda expression of lambda calculus and the ALGOL 68 routinetext. An abstraction is a construct like (con x: t_1; res y: t_2 | S), which can be considered as a statement S parameterized with regard to the variables x and y. We consider two kinds of parameters which are generally known as constant [Brinch Hansen] and result parameters [Wirth 2]. An abstraction may be applied to actual parameters of appropriate kinds. Such an application is a new form of statement; its meaning is that of a block obtained in a systematic way from the abstraction and the actual parameters.

In section 4.2.1 we define the syntax of abstractions and applications, partly by means of an attribute grammar and partly by means of the functions USE, ASSN, and INIT. In section 4.2.2 we define the semantics by means of the parameterized condition transformers pwp and pwlp. In section 4.2.3 we present some proof rules for applications and we prove their soundness.

4.2.1. Syntax

In this section we define the syntax of abstractions and applications. We do so by presenting some extensions to the attribute grammar for the kernel language given in section 2.3.2. These extensions do not yet completely define the syntax. An additional context condition will tentatively be expressed by means of the functions USE, ASSN, and INIT. Eventually it will be incorporated into the attribute grammar for the source language. The extensions follow:

Operations on *Names*, *Type*, and *Types*

Mts: *Names* * *Type* → *Types*

$\text{Mts}([n]_N, t) = [t]_T$

$\text{Mts}(ns_1 \textcircled{N} ns_2, t) = \text{Mts}(ns_1, t) \oplus \text{Mts}(ns_2, t)$.

{Informally, Mts(ns,t) yields a sequence the elements of which all equal t and the length of which equals the number of names in ns.}

Nonterminals

{Abstr <*Types*,*Types*>, Pdecs <*Decs*,*Types*>}

Terminals

{"con","res"}

Grammar rules

A1. Stat <e> ::= Abstr <ts_1,ts_2> (Exprs <e,ts_1> ;
Vars <e,ns,ts_2>) ■
($\underline{A}$ n: *Name* | $\#_N$ (n,ns) ≤ 1)

A2. Abstr <ts_1,ts_2> ::= (con Pdecs <d_1,ts_1> ; res Pdecs <d_2,ts_2>
| Stat <e_1>) ■
($\underline{A}$ n: *Name* | $\#_D$ (n,d_1 ⫽ d_2) ≤ 1)
e_1 = Ext(Empty,d_1 ⫽ d_2)

A3. Pdecs <d_0,ts_0> ::= Pdecs <d_1,ts_1> , Pdecs <d_2,ts_2> ■
$d_0 = d_1 \text{⫽} d_2$
$ts_0 = ts_1 \oplus_T ts_2$

A4. Pdecs <d,ts> ::= Ids <ns> : Type <t> ■
d = $[nd,t]_D$
ts = Mts(ns,t)

Explanation

An abstraction - i.e. an element of *L*(Abstr <*Types*,*Types*>) - is a construct of the form (con pd_1; res pd_2 | S), where pd_1 and pd_2 are parameter declarations - i.e. elements of *L*(Pdecs <*Decs*,*Types*>) - and S is a statement. In the parameter declarations the formal parameters and their types are listed. By means of the symbols "con" and "res" the formal parameters are classified as constant and result parameters, respectively. Their names must be mutually different. These names are the only variable names that may occur in S. In other words, S has no

access to nonlocal variables. Below we will formulate some additional restrictions. An abstraction may be applied to actual parameters of corresponding kind and type. An actual constant parameter is an expression; an actual result parameter is a variable. Actual result parameters must be mutually different.

□

Example

The following is an element of L(Stat <*Env*>) as defined by the extensions above:

(con b: bool,x,y: int; res z: int | if b → z := x-y [] ¬b → z := y-x fi)
(true,3,a+4;c)

□

Abstractions have to satisfy some additional conditions, viz. that the constituent statement does not assign to the constant parameters, and that it initializes the result parameters. Like we did for blocks, we will formulate these conditions in terms of the functions USE, ASSN, and INIT. Before we do so we have to extend the definitions of these functions to applications. As preparation for section 4.2.2 we also extend the definition of substitution.

Definition 4.13 {USE,ASSN,INIT}

For all $A(E_1,\ldots,E_m;v_1,\ldots,v_n) \in L$(Stat <*Env*>):

1. $USE(A(E_1,\ldots,E_m;v_1,\ldots,v_n)) = \bigcup_{i=1}^{m} USE(E_i)$.
2. $ASSN(A(E_1,\ldots,E_m;v_1,\ldots,v_n)) = \{v_1,\ldots,v_n\}$.
3. $INIT(A(E_1,\ldots,E_m;v_1,\ldots,v_n)) = \{v_1,\ldots,v_n\} \setminus \bigcup_{i=1}^{m} USE(E_i)$.

□

Definition 4.14 {(x ← y), substitution}

For all $x,y \in L$(Var <*Env*,*Name*,*Type*>),
$A(E_1,\ldots,E_m;v_1,\ldots,v_n) \in L$(Stat <*Env*>):

$$(x \leftarrow y)(A(E_1,\ldots,E_m;v_1,\ldots,v_n)) =$$
$$= A((x \leftarrow y)E_1,\ldots,(x \leftarrow y)E_m;(x \leftarrow y)v_1,\ldots,(x \leftarrow y)v_n) .$$

□

Note that no substitution inside the abstraction A is needed because its statement does not access global variables.

The additional context condition for abstractions is:

For each abstraction (con x_1: t_1,...,x_m: t_m; res y_1: t'_1,...,y_n: t'_n | S):

1. $\{x_1,\ldots,x_m\} \cap ASSN(S) = \emptyset$.
2. $\{y_1,\ldots,y_n\} = INIT(S)$.

4.2.2. Semantics

In this section we will define the semantics of abstractions and applications. For the sake of simplicity we will restrict ourselves to abstractions of the form (con x: t_1; res y: t_2 | S), i.e. to abstractions with one constant parameter and one result parameter only. Generalization to more parameters is straightforward.

It is our intention that the application

(con x: t_1; res y: t_2 | S)(E;v)

is semantically equivalent to the block

|[var x: t_1, y: t_2 | x := E; S; v := y]|

or rather, to the block

|[var x': t_1, y': t_2 | x' := E; S'; v := y']|

where x' and y' are fresh variables and S' = (x,y ← x',y')S.
{Note that this block satisfies the context conditions of section 4.1.1.}
This amounts to the identity

$$wp_e((\underline{con}\ x\colon t_1;\ \underline{res}\ y\colon t_2 \mid S)(E;v)) =$$
$$= (x' \leftarrow E) \circ wp_{e'}(S') \circ (v \leftarrow y') \ .$$

For the language considered thus far this identity could well serve as definition, but later one we will also encounter isolated occurrences of abstractions, as well as applications of one and the same abstraction to different actual parameters. To cope with such cases we must

define the semantics of an abstraction in isolation. Since an abstraction can be considered as a parameterized statement it will not come as a surprise that we define its semantics by means of a parameterized condition transformer. As preparation we give the following definition.

Definition 4.15 $\{P_e, \sqsubseteq_{P_e}\}$
For all e $\in$ *Env*:

1. $P_e = L(\text{Exprs } \langle e, Types\rangle) \times L(\text{Vars } \langle e, Names, Types\rangle) \to T_e$.
2. $\sqsubseteq_{P_e}$ is the standard order on P_e.

□

Theorem 4.16
For all e $\in$ *Env*: $(P_e, \sqsubseteq_{P_e})$ is a ccl.

□

Proof
Immediately by theorem 3.10.2 and theorem 3.36.

□

Definition 4.17 {pwp, pwlp}
For all e $\in$ *Env*, (con x: t_1; res y: t_2 | S) $\in L(\text{Abstr } \langle e, Types, Types\rangle)$:

Let $E = L(\text{Expr } \langle e, Prio, t_1\rangle)$,
$V = L(\text{Var } \langle e, Name, t_2\rangle)$,
$x', y' \in Name$ such that $\vdash \text{new}(x', e)$, $\vdash \text{new}(y', e)$, $x' \neq y'$,
$e' = \text{Ext}(e, [x', t_1]_D \textcircled{D} [y', t_2]_D)$,
$S' = (x, y \leftarrow x', y')S$.

The functions

$$pwp_e, pwlp_e \in L(\text{Abstr } \langle Env, Types, Types\rangle) \to P_e$$

are defined by:

1. $pwp_e((\text{con } x: t_1; \text{ res } y: t_2 \mid S)) =$
   $= (\lambda E \in E, v \in V \mid (x' \leftarrow E) \circ wp_e(S') \circ (v \leftarrow y))$.
2. $pwlp_e((\text{con } x: t_1; \text{ res } y: t_2 \mid S)) =$
   $= (\lambda E \in E, v \in V \mid (x' \leftarrow E) \circ wlp_e(S') \circ (v \leftarrow y'))$.

□

Definition 4.18 {wp, wlp for applications}
For all e $\in$ *Env*, A(E;v) $\in$ *L*(Stat <e>):

1. $wp_e(A(E;v)) = pwp_e(A)(E,v)$.
2. $wlp_e(A(E;v)) = pwlp_e(A)(E,v)$.

□

Note
As in section 4.1.3, we note that the theorems of sections 3.2.4, 3.3, and 4.1 also hold when the kernel language is extended with blocks and with applications as defined in this section. Proof omitted.
□

4.2.3. Proof rules

In this section we present some proof rules for applications and we prove their soundness. Given an abstraction

(con x: t_1; res y: t_2 | S)

and a correctness formula

x: t_1, y: t_2 | $[q_1(x)]$ S $[q_2(x,y)]$

we want to be able to derive correctness formulae for particular applications, e.g.

c | $[q_1(E)]$ (con x: t_1; res y: t_2 | S)(E;v) $[q_2(E,v)]$,

where the actual pre- and post-conditions $q_1(E)$ and $q_2(E,v)$ are obtained from the formal conditions $q_1(x)$ and $q_2(x,y)$ by substitution of the actual parameters E and v for the formal parameters x and y. Substitutions of this kind are not generally applicable, as the following counter example, adapted from [Hoare 3], shows:

Example
Consider the abstraction

(con x: int; res y: int | y := x+1) .

From the assignment axiom TA3 it follows that

$\vdash_{TC_0}$ x: int, y: int | [true] y := x+1 [y = x+1] .

For the application

(con x: int; res y: int | y := x+1)(z;z)

substitution would yield the correctness formula

c | [true] (con x: int; res y: int | y := x+1)(z;z) [z = z+1]

which is not a valid formula.
□

The problems are essentially due to the fact that substitution is not a reversible action. To avoid them we present two pairs of rules. One pair embodies the simple substitutions above but is only applicable under a certain disjointness condition for the actual parameters. The other pair is slightly more complex, but generally applicable. Soundness of both pairs can easily be proven.

Definition 4.19 {TR8, PR8, TR9, PR9}

For all e ∈ *Env* ,
(con x: t_1; res y: t_2 | S) ∈ L(Abstr <*Types*,*Types*>) ,
E ∈ L(Expr <e,*Prio*,t_1>) ,
v ∈ L(Var <e,*Name*,t_2>) ,
c ∈ L(Cont <e>) ,
C ∈ *Name* such that ⊢ new(C,e) ,
e_0 = Ext(Empty,$[x,t_1]_D$ ⓓ $[y,t_2]_D$) ,
q_1, q_2 ∈ L(Cond <e_0>) such that USE(q_1) ⊆ {x}
{below we write $q_1(x)$ and $q_2(x,y)$ to indicate upon which entities q_1 and q_2 depend}

the proof rules TR8, PR8, TR9, PR9 are defined by:

TR8. $$\frac{\text{x: } t_1,\ \text{y: } t_2 \mid [q_1(x)]\ S\ [q_2(x,y)]}{c \mid [q_1(E)]\ (\underline{\text{con}}\ \text{x: } t_1;\ \underline{\text{res}}\ \text{y: } t_2 \mid S)(E;v)\ [q_2(E,v)]}$$

provided v ∉ USE(E).

$$\text{PR8.}\ \frac{x\colon t_1,\ y\colon t_2 \mid \{q_1(x)\}\ S\ \{q_2(x,y)\}}{c \mid \{q_1(E)\}\ (\underline{con}\ x\colon t_1;\ \underline{res}\ y\colon t_2 \mid S)(E;v)\ \{q_2(E,v)\}}$$

provided $v \notin USE(E)$.

$$\text{TR9.}\ \frac{x\colon t_1,\ y\colon t_2 \mid [q_1(x)]\ S\ [q_2(x,y)]}{c \triangleright C\colon t_1 \mid [E{=}C \wedge q_1(C)]\ (\underline{con}\ x\colon t_1;\ \underline{res}\ y\colon t_2 \mid S)(E;v)\ [q_2(C,v)]}$$

$$\text{PR9.}\ \frac{x\colon t_1,\ y\colon t_2 \mid \{q_1(x)\}\ S\ \{q_2(x,y)\}}{c \triangleright C\colon t_1 \mid \{E{=}C \wedge q_1(C)\}\ (\underline{con}\ x\colon t_1;\ \underline{res}\ y\colon t_2 \mid S)(E;v)\ \{q_2(C,v)\}}$$

□

Theorem 4.20

1. Tsound0 (TR8) .
2. Psound0 (PR8) .
3. Tsound0 (TR9) .
4. Psound0 (PR9) .

□

Proof

We give a combined proof of 1 and 3. The proof of 2 and 4 is similar.

Let $x',y' \in \underline{Name}$ such that $\vdash new(x',e)$, $\vdash new(y',e)$, $x' \neq y'$, $x' \neq C$, $y' \neq C$.

Let $e_0' = Ext(Empty,[x',t_1]_D \textcircled{D} [y',t_2]_D)$,
$e_1 = Ext(e,[C,t_1]_D)$,
$e_2 = Ext(e_1,[x',t_1]_D \textcircled{D} [y',t_2]_D)$.

Let $X \in L(Expr\ \langle e_1, Prio, t_1\rangle)$ such that $v \notin USE(X)$,
$S' = (x,y \leftarrow x',y')S$.

$Tvalid0(x\colon t_1,\ y\colon t_2 \mid [q_1(x)]\ S\ [q_2(x,y)])$

= {definition 3.48.2.2, definition 3.33}

$q_1(x) \sqsubseteq_{e_0} wp_{e_0}(S)q_2(x,y)$

=

$q_1(x') \sqsubseteq_{e_0'} wp_{e_0'}(S')q_2(x',y')$

$\Rightarrow$

(1) $\quad q_1(x') \sqsubseteq_{e_2} wp_{e_2}(S')q_2(x',y')$.

Consider

$$
\begin{aligned}
& wp_{e_1}((\underline{\text{con}}\ x\colon t_1;\ \underline{\text{res}}\ y\colon t_2 \mid S)(E;v))q_2(X,v) \\
= & \ \{\text{definitions 4.18.1, 4.17.1}\} \\
& (x' \leftarrow E)wp_{e_2}(S')(v \leftarrow y')q_2(X,v) \\
= & \ \{v \notin USE(X),\ \text{lemma 4.7, definition 4.6.10.1}\} \\
& (x' \leftarrow E)wp_{e_2}(S')q_2(X,y') \\
\sqsupseteq_{e_2} & \ \{\text{corollary 3.40.1}\} \\
& (x' \leftarrow E)wp_{e_2}(S')(x' = X \wedge q_2(x',y')) \\
= & \ \{ASSN(S') \cap USE(x' = X) = \emptyset,\ \text{lemma 4.5.5}\} \\
& (x' \leftarrow E)(x' = X \wedge wp_{e_2}(S')q_2(x',y')) \\
\sqsupseteq_{e_2} & \ \{(1),\ \text{theorem 3.42.1}\} \\
& (x' \leftarrow E)(x' = X \wedge q_1(x')) \\
= & \ \{x' \notin USE(X),\ \text{lemma 4.7}\} \\
& E = X \wedge q_1(E) \ .
\end{aligned}
$$

Under the assumption that

$$Tvalid0(x\colon t_1,\ y\colon t_2 \mid [q_1(x)]\ S\ [q_2(x,y)])$$

holds, we have proven

$$E = X \wedge q_1(E) \sqsubseteq_{e_2} wp_{e_1}((\underline{\text{con}}\ x\colon t_1,\ \underline{\text{res}}\ y\colon t_2 \mid S)(E;v))q_2(X,v)$$

from which follows

(2) $\quad \vdash_D c \triangleright C\colon t_1 \mid (E=X) \wedge q(E)$

$$\Rightarrow wp_{e_1}((\underline{\text{con}}\ x\colon t_1,\ \underline{\text{res}}\ y\colon t_2 \mid S)(E;v))q_2(X,v) \ .$$

We are still free to choose X, subject to the condition $v \notin USE(X)$. Two choices are of interest:

a. If we choose X = E, (2) can be simplified to

$\vdash_D c \mid q_1(E) \Rightarrow wp_e((\underline{con}\ x\colon t_1, \underline{res}\ y\colon t_2 \mid S)(E;v))q_2(E,v)$,

i.e.

$Tvalid0(c \mid [q_1(E)]\ (\underline{con}\ x\colon t_1, \underline{res}\ y\colon t_2 \mid S)(E;v)\ [q_2(E,v)])$

provided $v \notin USE(E)$, hence Tsound0 (TR8).

b. If we choose X = C, $v \notin USE(X)$ is satisfied, and we obtain from (2)

$\vdash_D c \triangleright C\colon t_1 \mid (E=C) \wedge q_1(E)$

$\Rightarrow wp_{e_1}((\underline{con}\ x\colon t_1, \underline{res}\ y\colon t_2 \mid S)(E;v))q_2(C,v)$

or equivalently:

$\vdash_D c \triangleright C\colon t_1 \mid (E=C) \wedge q_1(C)$

$\Rightarrow wp_{e_1}((\underline{con}\ x\colon t_1, \underline{res}\ y\colon t_2 \mid S)(E;v))q_2(C,v)$,

i.e.

$Tvalido(c \triangleright C\colon t_1$

$\mid [E=C \wedge q_1(C)]\ (\underline{con}\ x\colon t_1, \underline{res}\ y\colon t_2 \mid S)(E;v)\ [q_2(C,v)])$,

hence Tsound0 (TR9).

□

4.3. Parameterless recursive procedures

4.3.0. Introduction

The third aspect of the procedure concept we want to study in isolation is recursion. In order to concentrate on this subject we will temporarily - i.e. throughout section 4.3 - ignore other aspects of the language definition, such as precise syntax, scope rules, nested declarations, parameter mechanisms, etc. We stress the point that the constructs considered in this section are not part of the source language. They only serve to study the effect of recursion on the structure of semantic equations, proof rules, and soundness proofs. Accordingly, the scope of definitions and theorems concerning these constructs is limited to section 4.3.

We will study programs of the form

$$p_1 = S_1, \ldots, p_k = S_k \mid S$$

where the constructs $p_i = S_i$ (i: $1 \leq i \leq k$) are declarations of parameterless and possibly recursive procedures. We omit a precise specification of the syntax of these programs. Suffice it to say that the names p_i are elements of *Name*, that they are mutually different and also different from variable names, that they may occur as statement in $S_1, \ldots, S_k, S$, and that the statements $S_1, \ldots, S_k, S$ contain neither blocks nor abstractions. We omit the specification of environments for wp, wlp, C, T, etc.

In section 4.3.1 we define the semantics of these programs. In section 4.3.2 we present some proof rules and prove their soundness.

4.3.1. Semantics

It is our intention, roughly speaking, that a statement p_i has the same condition transformer as the corresponding statement S_i. The condition transformer of a statement should therefore be defined relatively to a set of procedure declarations. This can be achieved by means of an extra argument δ for wp and wlp, being a mapping from procedure names p_i to condition transformers of the corresponding

statements S_i. As the procedures may be mutually recursive the δ corresponding to a set of procedure declarations will be determined by means of fixed point techniques. The intentions just sketched are captured by definitions 4.21, 4.23/24, 4.26. In their structure these definitions much resemble those encountered in denotational semantics; see e.g. [de Bakker].

Definition 4.21 {Δ}

1. $\Delta = \mathit{Name} \to T$.
2. $\sqsubseteq_\Delta$ is the standard order on Δ.

□

Theorem 4.22

$(\Delta, \sqsubseteq_\Delta)$ is a ccl.

□

Proof

Immediately by theorem 3.10.2 and theorem 3.36.

□

As already said the definitions of wp and wlp will be extended with an argument $\delta \in \Delta$. For the statements of the kernel language this argument is largely ignored, but it is essential for the procedure statements. We will denote the extended versions by wp' and wlp', respectively.

Definition 4.23 {wp'}

The function $wp' \in L(Stat) \to (\Delta \to T)$ is defined by:

1. $wp'(abort)\delta = (\lambda q \in C \mid false)$.
2. $wp'(skip)\delta = (\lambda q \in C \mid q)$.
3. $wp'(v := E)\delta = (v \leftarrow E)$.
4. $wp'(S_1;S_2)\delta = (wp'(S_1)\delta) \circ (wp'(S_2)\delta)$.
5. $wp'(\underline{if}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{fi})\delta =$
   $(\lambda q \in C \mid [\bigvee i \mid 1 \le i \le n \mid B_i] \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow (wp'(S_i)\delta)q])$.
6. $wp'(\underline{do}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{od})\delta = \mu F$,

where $F = (\lambda f \in C \to_{uc} C \mid (\lambda q \in C \mid ([\bigvee i \mid 1 \le i \le n \mid B_i] \vee q) \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow (wp'(S_i)\delta)f\ q]))$.

7. $wp'(p)\delta = \delta(p)$.

□

Definition 4.24 {wlp'}

The function $wlp' \in L(Stat) \to (\Delta \to T)$ is defined by:

1. $wlp'(abort)\delta = (\lambda q \in C \mid true)$.
2. $wlp'(skip)\delta = (\lambda q \in C \mid q)$.
3. $wlp'(v := E)\delta = (v \leftarrow E)$.
4. $wlp'(S_1;S_2)\delta = (wlp'(S_1)\delta) \circ (wlp'(S_2)\delta)$.
5. $wlp'(\underline{if}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{fi})\delta =$
   $(\lambda q \in C \mid [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow (wlp'(S_i)\delta)q])$.
6. $wlp'(\underline{do}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{od})\delta = \nu G$,
   where $G = (\lambda f \in C \to_{dc} C \mid (\lambda q \in C \mid ([\bigvee i \mid 1 \le i \le n \mid B_i] \vee q) \wedge [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow (wlp'(S_i)\delta)f\ q]))$.
7. $wlp'(p)\delta = \delta(p)$.

□

As with definitions 3.37 and 3.38 in chapter 3, definitions 4.23 and 4.24 above are well-formed only if F and G are continuous, which in turn depends on continuity of wp' and wlp'. That this is indeed the case is stated in the following theorem.

Theorem 4.25

1. For all $S \in L(Stat)$, $\delta \in \mathit{Name} \to (C \to_{uc} C)$:

   $wp'(S)\delta \in C \to_{uc} C$.

   If S is of the form $\underline{do}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{od}$, and F is as in definition 4.23.6, then

   $F \in (C \to_{uc} C) \to_{uc} (C \to_{uc} C)$.

2. For all $S \in L(\text{Stat})$, $\delta \in \underline{Name} \to (C \to_{dc} C)$:

$$wlp'(S)\delta \in C \to_{dc} C .$$

If S is of the form $\underline{do}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{od}$, and G is as in definition 4.24.6, then

$$G \in (C \to_{dc} C) \to_{dc} (C \to_{dc} C) .$$

□

Proof

We only consider 1.

The proof is by induction on the composition of S. Apart from the δ argument the first six cases are identical to those in the proof of theorem 3.39.1. Therefore we only consider the case that S is a procedure statement.

1.7. $\underline{S :: p}$

$$\begin{aligned} & wp'(S)\delta \\ = \ & \{\text{definition 4.23.7}\} \\ & \delta(p) \\ \in \ & \{\delta \in \underline{Name} \to (C \to_{uc} C)\} \\ & C \to_{uc} C . \end{aligned}$$

□

For a program $p_1 = S_1, \ldots, p_k = S_k \mid S$ we will define the condition transformers wp and wlp in terms of wp' and wlp' respectively, where the argument $\delta \in \Delta$ depends on the procedure declaration part.

Definition 4.26 {wp and wlp for $p_1 = S_1, \ldots, p_k = S_k \mid S$}

1. $wp(p_1 = S_1, \ldots, p_k = S_k \mid S) = wp'(S)\delta$,

where δ is the function $\{(p_1,\varphi_1), \ldots, (p_k,\varphi_k)\}$, $(\varphi_1, \ldots, \varphi_k) = \mu(\Phi_1, \ldots, \Phi_k)$, and, for i: $1 \le i \le k$:

$$\Phi_i = (\lambda\varphi_1', \ldots, \varphi_k' \in C \to_{uc} C \mid wp'(S_i)\{(p_1,\varphi_1'), \ldots, (p_k,\varphi_k')\}) .$$

2. $wlp(p_1 = S_1,\ldots,p_k = S_k \mid S) = wlp'(S)\delta$,

where δ is the function $\{(p_1,\psi_1),\ldots,(p_k,\psi_k)\}$,
$(\psi_1,\ldots,\psi_k) = \nu(\Psi_1,\ldots,\Psi_k)$, and, for i: $1 \le i \le k$:

$$\Psi_i = (\lambda\psi_1',\ldots,\psi_k' \in C \to_{dc} C \mid wlp'(S_i)\{(p_1,\psi_1'),\ldots,(p_k,\psi_k')\}) .$$

□

Again, these definitions are well-formed only if the functions Φ_i and Ψ_i are upward continuous and downward continuous, respectively. This is assured by the following theorem:

Theorem 4.27

For all $S \in L(Stat)$:

1. $(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid wp'(S)\{(p_1,\varphi_1),\ldots,(p_k,\varphi_k)\})$

$\in (C \to_{uc} C)^k \to_{uc} (C \to_{uc} C)$.

2. $(\lambda\psi_1,\ldots,\psi_k \in C \to_{dc} C \mid wlp'(S)\{(p_1,\psi_1),\ldots,(p_k,\psi_k)\})$

$\in (C \to_{dc} C)^k \to_{dc} (C \to_{dc} C)$.

□

Proof

We only prove 1; the proof of 2 is similar when dual versions of theorems 3.24 and 3.26 are used.

The proof is by induction on the composition of S. In the scope of $\varphi_1,\ldots,\varphi_k \in C \to_{uc} C$ we will write δ for $\{(p_1,\varphi_1),\ldots,(p_k,\varphi_k)\}$.

1.1 - 1.3. S :: abort, S :: skip, S :: v := E

By definition 4.23.1-3:
for all $\varphi_1,\ldots,\varphi_k \in C \to_{uc} C$: $wp'(S)\delta \in C \to_{uc} C$ and independent of $\varphi_1,\ldots,\varphi_k$.
So, by theorem 3.24.1:

$$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid wp'(S)\delta) \in (C \to_{uc} C)^k \to_{uc} (C \to_{uc} C) .$$

1.4. $S :: S_1;S_2$

H. For all i: $1 \le i \le 2$:

$$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid wp'(S_i)\delta) \in (C \to_{uc} C)^k \to_{uc} (C \to_{uc} C) .$$

$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid wp'(S)\delta)$

= {definition 4.23.4}

$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid (wp'(S_1)\delta) \circ (wp'(S_2)\delta))$

$\in$ {H, theorem 3.26.3}

$(C \to_{uc} C)^k \to_{uc} (C \to_{uc} C)$.

1.5. $S :: \underline{\text{if}}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{\text{fi}}$

H. For all i: $1 \le i \le n$:

$$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid wp'(S_i)\delta) \in (C \to_{uc} C)^k \to_{uc} (C \to_{uc} C) .$$

$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid wp'(S)\delta)$

= {definition 4.23.5}

$$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid$$
$$(\lambda q \in C \mid [\bigvee i \mid 1\le i\le n \mid B_i]$$
$$\wedge\, [\bigwedge i \mid 1\le i\le n \mid \neg B_i \vee (wp'(S_i)\delta)q]$$
$$)$$
$$)$$

$\in$ {H, repeated application of theorem 3.26}

$(C \to_{uc} C)^k \to_{uc} (C \to_{uc} C)$.

1.6. $S :: \underline{\text{do}}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{\text{od}}$

H. For all i: $1 \le i \le n$:

$$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid wp'(S_i)\delta) \in (C \to_{uc} C)^k \to_{uc} (C \to_{uc} C) .$$

We have to prove continuity of

$$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid \mu(\lambda f \in T \mid (\lambda q \in C \mid A(\varphi_1,\ldots,\varphi_k)))) ,$$

where $A(\varphi_1,\ldots,\varphi_k)$ stands for

$$([\bigvee i \mid 1\le i\le n \mid B_i] \vee q)$$
$$\wedge\ [\bigwedge i \mid 1\le i\le n \mid B_i \Rightarrow (wp'(S)\{(p_1,\varphi_1),\ldots,(p_k,\varphi_k)\})\ fq]) .$$

Let $\langle\bar{\varphi}_j\rangle_{j=0}^{\infty}$ be an ascending chain in $(C \to_{uc} C)^k$.

$$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid \mu(\lambda f \in T \mid (\lambda q \in C \mid A(\varphi_1,\ldots,\varphi_k))))(\sqcup_{j=0}^{\infty} \bar{\varphi}_j)$$
$$= \{\beta\text{-reduction}\}$$
$$\mu(\lambda f \in T \mid (\lambda q \in C \mid A(\sqcup_{j=0}^{\infty} \bar{\varphi}_j)))$$
$$= \{H, \text{repeated application of 3.26}\}$$
$$\mu(\lambda f \in T \mid (\lambda q \in C \mid \sqcup_{j=0}^{\infty} A(\bar{\varphi}_j)))$$
$$= \{\text{definition lub}\}$$
$$\mu(\lambda f \in T \mid \sqcup_{j=0}^{\infty} (\lambda q \in C \mid A(\bar{\varphi}_j)))$$
$$= \{\text{definition lub}\}$$
$$\mu(\sqcup_{j=0}^{\infty} (\lambda f \in T \mid (\lambda q \in C \mid A(\bar{\varphi}_j))))$$
$$= \{\text{continuity } \mu, \text{ see [de Bakker, theorem 5.11]}\}$$
$$\sqcup_{j=0}^{\infty} \mu(\lambda f \in T \mid (\lambda q \in C \mid A(\bar{\varphi}_j)))$$
$$= \{\beta\text{-expansion}\}$$
$$\sqcup_{j=0}^{\infty} (\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid (\lambda f \in T \mid (\lambda q \in C \mid A(\varphi_1,\ldots,\varphi_k))))(\bar{\varphi}_j) .$$

1.7. $S ::= p_j$

$$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid wp'(S)\delta)$$
$$= \{\text{definition 4.23.7}\}$$
$$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid \delta(p_j))$$
$$= \{\delta = \{(p_1,\varphi_1),\ldots,(p_k,\varphi_k)\}\}$$
$$(\lambda\varphi_1,\ldots,\varphi_k \in C \to_{uc} C \mid \varphi_j)$$
$$\in \{\text{theorem 3.24.2}\}$$
$$(C \to_{uc} C)^k \to_{uc} (C \to_{uc} C) .$$

□

As already said it is our intention that a statement p_i has the same condition transformer as the corresponding statement S_i. That such is indeed the case is stated in the following theorem.

Theorem 4.28

Let $p_1 = S_1, \ldots, p_k = S_k \mid S$ be a program.

1. Let δ be as in definition 4.26.1.
   For all i: $1 \leq i \leq k$: $wp'(p_i)\delta = wp'(S_i)\delta$.

2. Let δ be as in definition 4.26.2.
   For all i: $1 \leq i \leq k$: $wlp'(p_i)\delta = wlp'(S_i)\delta$.

□

Proof

We only prove 1; the proof of 2 is similar.

Let i: $1 \leq i \leq k$.

$wp'(p_i)\delta$

= {definition 4.23.7}

$\delta(p_i)$

= {definition 4.26.1, δ}

φ_i

= {fixed point propery}

$\Phi_i(\varphi_1, \ldots, \varphi_k)$

= {definition 4.26.1, Φ_i}

$wp'(S_i)\delta$.

□

The following examples illustrate how certain recursive procedures relate to statements of the kernel language.

Examples

1. $wp(p = p \mid p)$

$= \{\text{definition } 4.26.1\}$

$wp'(p)\{(p,\varphi)\}$

$= \{\text{definition } 4.23.7\}$

φ

$= \{\text{definition } 4.26.1\}$

$\mu(\lambda\varphi' \mid wp'(p)\{(p,\varphi')\})$

$= \{\text{definition } 4.23.7\}$

$\mu(\lambda\varphi' \mid \varphi')$

$= (\lambda q \mid \text{false})$

$= \{\text{definition } 4.23.1\}$

$wp'(\text{abort})\{(p,\varphi)\}$.

2. $wp(p = \underline{\text{if}}\ B_1 \to S_1;\ p\ \square \ldots \square\ B_n \to S_n;\ p$
$\square\ \neg [\bigvee i \mid 1 \le i \le n \mid B_i] \to \text{skip}\ \underline{\text{fi}} \mid p)$

$= \{\text{definition } 4.26.1\}$

$wp'(p)\{(p,\varphi)\}$

$= \{\text{definition } 4.23.7\}$

φ

$= \{\text{definition } 4.26.1\}$

$\mu(\lambda\varphi' \mid wp'(\underline{\text{if}} \ldots \underline{\text{fi}})\{(p,\varphi')\})$

$= \mu(\lambda\varphi' \mid (\lambda q \mid ([\bigvee i \mid 1 \le i \le n \mid B_i] \vee \neg [\bigvee i \mid 1 \le i \le n \mid B_i])$
$\wedge [\bigwedge i \mid B_i \Rightarrow wp'(S_i;p)\{(p,\varphi')\}q]$
$\wedge (\neg [\bigvee i \mid 1 \le i \le n \mid B_i] \Rightarrow wp'(\text{skip})\{(p,\varphi')\}q)))$

$= \{\text{definitions } 4.23.4, 4.23.7, 4.23.2, \text{prop. log.}\}$

$\mu(\lambda\varphi' \mid (\lambda q \mid [\bigwedge i \mid 1 \le i \le n \mid B_i \Rightarrow wp'(S_i)\{(p,\varphi')\}\varphi' q]$
$\wedge ([\bigvee i \mid 1 \le i \le n \mid B_i] \vee q)))$

$= \{\varphi = \mu(\lambda\varphi' \mid g(\varphi',\varphi'))$ is equivalent to $\varphi = \mu(\lambda\varphi' \mid g(\varphi,\varphi'))$, see [de Bakker, p. 141]$\}$

$$\mu(\lambda\varphi' \mid (\lambda q \mid [\wedge i \mid 1\leq i\leq n \mid B_i \Rightarrow wp'(S_i)\{(p,\varphi)\}\,\varphi' q] \wedge ([\vee i \mid 1\leq i\leq n \mid B_i] \vee q)))$$

$= \{$definition 4.23.6$\}$

$$wp'(\underline{do}\ B_1 \rightarrow S_1 \,[]\, \ldots \,[]\, B_n \rightarrow S_n\ \underline{od})\{(p,\varphi)\}$$

□

Note

As the derivation in the second example is independent of the structure of the statements S_i, it follows that the stated equivalence also holds in cases where the S_i contain occurrences of p.

□

4.3.2. Proof rules

In this section we study proof rules for programs involving parameterless recursive procedures. As already mentioned we ignore syntactic issues etc. Since statements containing procedure variables can only be interpreted relatively to a set of procedure declarations the notions of validity and soundness have to be redefined, which forces us to reconsider the axioms and proof rules of PC_0 and TC_0. Apart from these we will study two kinds of proof rules: for nonrecursive procedures simple rules relating a correctness formula for S_i to one for the corresponding p_i suffice; for recursive procedures certain induction rules are required. As far as the treatment of induction is concerned there is an essential difference between the partial and the total correctness cases. The partial correctness induction rule is based on greatest fixed point induction, whereas the total correctness induction rule is based on mathematical induction and the fixed point property. The two cases are treated in sections 4.3.2.1 and 4.3.2.2, respectively. Section 4.3.2.3 contains a short comparison of the two induction rules.

4.3.2.1. Proof rules for partial correctness

In this section we discuss the partial correctness logic for programs involving parameterless recursive procedures. We begin with definitions of validity and soundness.

Definition 4.29 {Pvalid1}
On "partial correctness formulae" × "procedure declarations" the predicate Pvalid1 is defined as:

1. $\text{Pvalid1}(c \mid q, (p_1=S_1,\ldots,p_n=S_n)) = \vdash_D c \mid q$.

2. $\text{Pvalid1}(c \mid \{q_1\}\ S\ \{q_2\}, (p_1=S_1,\ldots,p_n=S_n)) =$

$$\vdash_D c \mid q_1 \Rightarrow \text{wlp'}(S)\{(p_1,\phi_1),\ldots,(p_n,\phi_n)\}\ q_2 ,$$

where $(\phi_1,\ldots,\phi_n) = \nu(\Psi_1,\ldots,\Psi_n)$ and $\Psi_1,\ldots,\Psi_n$ are as in definition 4.26.2.

□

Definition 4.30 {Psound1}
On "partial correctness proof rules" × "procedure declarations" the predicate Psound1 is defined as:

$$\text{Psound1}\left(\frac{f_1,\ldots,f_n}{g}, pd\right) =$$

$$(\underline{A}\ i \mid 1\le i\le n \mid \text{Pvalid1}(f_i,pd)) \Rightarrow \text{Pvalid1}(g,pd) .$$

□

The following theorem states that the partial correctness logic PC_0 of chapter 3 is also sound with respect to these revised definitions:

Theorem 4.31

1. $(\underline{A}\ a \in Ax_{PC_0} : \text{Pvalid1}(a, (p_1=S_1,\ldots,p_n=S_n)))$.

2. $(\underline{A}\ r \in Pr_{PC_0} : \text{Psound1}(r, (p_1=S_1,\ldots,p_n=S_n)))$.

□

Proof
Similar to that of theorem 3.52. Details omitted.
□

In addition ro Pr_{PC_0} we need rules to derive properties of procedure statements. The following theorem states the soundness of a rule for use with nonrecursive procedures.

Theorem 4.32

$$(\underline{A}\ i \mid 1 \leq i \leq n \mid \text{Psound1}\left(\frac{c \mid \{q_1\}\ S_i\ \{q_2\}}{c \mid \{q_1\}\ p_i\ \{q_2\}}\ ,\ (p_1=S_1,\ldots,p_n=S_n)\right)) .$$

□

Proof

Immediately by definitions 4.29 and 4.30 and by theorem 4.28.2.

□

For recursive procedures this rule is insufficient; some form of induction is required. In the literature [e.g. Hoare 3, Apt 1] an induction rule of the following form is sometimes used:

$$\frac{c \mid \{q_1\}\ p\ \{q_2\} \vdash_{PC_0} c \mid \{q_1\}\ S\ \{q_2\}}{c \mid \{q_1\}\ p\ \{q_2\}} .$$

It should be noted that the form of this rule is misleading. Actually it is not a proof rule since its premiss is not a correctness formula but a meta-statement about the derivability of one correctness formula from another one by means of the axioms and proof rules of PC_0 (see e.g. [Apt 2] for a discussion). Nevertheless we will adopt the above form, but we will interpret it as the following theorem.

Theorem 4.33

If $\quad c \mid \{q_1\}\ p\ \{q_2\} \vdash_{PC_0} c \mid \{q_1\}\ S\ \{q_2\}$

then Pvalid1$(c \mid \{q_1\}\ p\ \{q_2\},\ p = S)$.

□

Note

For simplicity we restrict ourselves to a single procedure. The results can easily be extended to programs with more procedures.

□

The proof of this theorem requires some extra provisions. The conclusion of the theorem is based on the interpretation of the condition transformer of p as $\nu\Psi$, viz. $\vdash_D c \mid q_1 \Rightarrow (\nu\Psi)q_2$. In order to prove this result by greatest fixed point induction we have to show that

$$(\underline{A}\ \phi \mid (\vdash_D c \mid q_1 \Rightarrow \phi q_2) \Rightarrow (\vdash_D c \mid q_1 \Rightarrow \Psi(\phi)q_2)) .$$

If we want to derive this from the premiss of the theorem it follows that we also have to consider soundness of PC_0 with respect to other interpretations ϕ. Therefore, in order to prove theorem 4.33 we first introduce different notions os validity and soundness and we show that PC_0 is sound in terms of these notions as well.

Definition 4.34 {Pvalid2}

On "partial correctness formulae" $\times$ $(C \rightarrow_{dc} C)$ the predicate Pvalid2 is defined as:

1. $\text{Pvalid2}(c \mid q,\phi) = \vdash_D c \mid q$.
2. $\text{Pvalid2}(c \mid \{q_1\}\ S\ \{q_2\},\phi) = \vdash_D c \mid q_1 \Rightarrow \text{wlp}'(S)\{(p,\phi)\}q_2$.

□

Definition 4.35 {Psound2}

On "partial correctness proof rules" $\times$ $(C \rightarrow_{dc} C)$ the predicate Psound2 is defined as:

$$\text{Psound2}\left(\frac{f_1,\ldots,f_n}{g},\ \phi\right) =$$

$$(\underline{A}\ i \mid 1 \le i \le n \mid \text{Pvalid2}(f_i,\phi)) \Rightarrow \text{Pvalid2}(g,\phi) .$$

□

Theorem 4.36

1. $(\underline{A}\ a \in Ax_{PC_0},\ \phi \in C \rightarrow_{dc} C:\ \text{Pvalid2}(a,\phi))$.
2. $(\underline{A}\ r \in Pr_{PC_0},\ \phi \in C \rightarrow_{dc} C:\ \text{Psound2}(r,\phi))$.

□

Proof

Similar to that of theorem 3.52. Details omitted.

□

Definitions 4.34 and 4.35 and theorem 4.36 enable us to prove theorem 4.33:

Proof of theorem 4.33

$c \mid \{q_1\}\ p\ \{q_2\} \vdash_{PC_0} c \mid \{q_1\}\ S\ \{q_2\}$

$\Rightarrow$ {theorem 4.36}

$(\underline{A}\ \psi \mid \mathrm{Pvalid2}(c \mid \{q_1\}\ p\ \{q_2\}, \psi) \Rightarrow \mathrm{Pvalid2}(c \mid \{q_1\}\ S\ \{q_2\}, \psi))$

= {definition 4.34}

$(\underline{A}\ \psi \mid (\vdash_D c \mid q_1 \Rightarrow \mathrm{wlp'}(p)\{(p,\psi)\}q_2)$

$\Rightarrow$

$(\vdash_D c \mid q_1 \Rightarrow \mathrm{wlp'}(S)\{(p,\psi)\}q_2)$

$)$

= {definition 4.23.7, definition 4.26.2, Ψ}

$(\underline{A}\ \psi \mid (\vdash_D c \mid q_1 \Rightarrow \psi q_2) \Rightarrow (\vdash_D c \mid q_1 \Rightarrow \Psi(\psi)q_2))$

$\{\vdash_D c \mid q_1 \Rightarrow (\lambda q \mid \mathrm{true})q_2$, g.f.p. induction, admissibility is trivial$\}$

$\vdash_D c \mid q_1 \Rightarrow (\nu\Psi)q_2$

= {definition 4.29}

$\mathrm{Pvalid1}(c \mid \{q_1\}\ p\ \{q_2\},\ p = S)$.

□

4.3.2.2. Proof rules for total correctness

The structure of this section parallels that of 4.3.2.1. We begin with definitions of validity and soundness relative to a set of parameterless recursive procedures:

Definition 4.37 {Tvalid1}

On "total correctness formulae" × "procedure declarations" the predicate Tvalid1 is defined as:

1. $\mathrm{Tvalid1}(c \mid q,\ (p_1=S_1,\ldots,p_n=S_n)) = \vdash_D c \mid q$.

2. $\mathrm{Tvalid1}(c \mid [q_1]\ S\ [q_2],\ (p_1{=}S_1,\ldots,p_n{=}S_n)) =$

$$\vdash_D c \mid q_1 \Rightarrow wp'(S)\{(p_1,\varphi_1),\ldots,(p_n,\varphi_n)\}q_2$$

where $(\varphi_1,\ldots,\varphi_n) = \mu(\Phi_1,\ldots,\Phi_n)$ and $\Phi_1,\ldots,\Phi_n$ are as in definition 4.26.1.

□

Definition 4.38 {Tsound1}

On "total correctness proof rules" × "procedure declarations" the predicate Tsound1 is defined as:

$$\mathrm{Tsound1}\left(\frac{f_1,\ldots,f_n}{g}\ ,\ pd\right) =$$

$$(\underline{A}\ i \mid 1{\le}i{\le}n \mid \mathrm{Tvalid1}(f_i,pd)) \Rightarrow \mathrm{Tvalid1}(g,pd)\ .$$

□

The following theorem states that the total correctness logic TC_0 of chapter 3 is also sound with respect to these revised definitions:

Theorem 4.39

1. $(\underline{A}\ a \in Ax_{TC_0} \mid \mathrm{Tvalid1}(a,\ (p_1{=}S_1,\ldots,p_n{=}S_n)))$.

2. $(\underline{A}\ r \in Pr_{TC_0} \mid \mathrm{Tsound1}(r,\ (p_1{=}S_1,\ldots,p_n{=}S_n)))$.

□

Proof

Similar to that of theorem 3.54. Details omitted.

□

For nonrecursive procedures we have the following analogue of theorem 4.32:

Theorem 4.40

$$(\underline{A}\ i \mid 1{\le}i{\le}n \mid \mathrm{Tsound1}\left(\frac{c \mid [q_1]\ S_i\ [q_2]}{c \mid [q_1]\ p_i\ [q_2]}\ ,\ (p_1{=}S_1,\ldots,p_n{=}S_n)\right))$$

□

Proof

Immediately by definitions 4.37 and 4.38 and by theorem 4.28.1.

□

Again we need an induction rule to deal with recursive procedures. The induction principle employed differs from that in section 4.3.2.1 however. There we had to use fixed point induction and to extend the notions of validity and soundness. Here ordinary mathematical induction is sufficient to prove the following theorem.

Theorem 4.41

Let $\langle q_i \rangle_{i=0}^{\infty}$ be a sequence of conditions.

If $(\underline{A}\ i \mid 0 \leq i \mid c \mid [[\bigvee j \mid 0 \leq j < i \mid q_j]]\ p\ [r] \vdash_{TC_0} c \mid [q_i]\ S\ [r])$

then $Tvalid1(c \mid [[\bigvee i \mid 0 \leq i \mid q_i]]\ p\ [r],\ p = S)$.

□

Note

As in section 4.3.2.1, we have restricted ourselves to programs with a single procedure.

□

Proof

$(\underline{A}\ i \mid 0 \leq i \mid c \mid [[\bigvee j \mid 0 \leq j < i \mid q_j]]\ p\ [r]$

$\vdash_{TC_0}$

$c \mid [q_i]\ S\ [r]$

)

⇒ {theorem 4.39}

$(\underline{A}\ i \mid 0 \leq i \mid Tvalid1(c \mid [[\bigvee j \mid 0 \leq j < i \mid q_j]]\ p\ [r],\ p = S)$

⇒

$Tvalid1(c \mid [q_i]\ S\ [r],\ p = S)$

)

= {definitions 4.37, 3.33, φ and Φ as in definition 4.26.1}

$$
\begin{array}{l}
\quad (\underline{A}\ i \mid 0 \le i \mid [\vee j \mid 0 \le j < i \mid q_j] \sqsubseteq \varphi r \\
\qquad\qquad \Rightarrow \\
\qquad\qquad q_i \sqsubseteq \Phi(\varphi) r \\
\quad) \\
= \{\varphi = \Phi(\varphi)\} \\
\quad (\underline{A}\ i \mid 0 \le i \mid [\vee j \mid 0 \le j < i \mid q_j] \sqsubseteq \varphi r \\
\qquad\qquad \Rightarrow \\
\qquad\qquad q_i \sqsubseteq \varphi r \\
\quad) \\
\Rightarrow \{\text{lemma 3.29}\} \\
\quad [\vee i \mid 0 \le i \mid q_i] \sqsubseteq \varphi r \\
= \{\text{definitions 4.37, 3.33, } \varphi\} \\
\quad \text{Tvalid1}(c \mid [[\vee i \mid 0 \le i \mid q_i]]\ p\ [r],\ p = S)\ .
\end{array}
$$

□

From theorem 4.41 a total correctness induction rule can easily be derived. If we define q_i as $q \wedge e = I$, where e is an integer expression and I a symbol sequence representing i, we obtain that

$$[\vee j \mid 0 \le j < i \mid q_j] = q \wedge 0 \le e \wedge e < I\ ,$$

and $[\vee i \mid 0 \le i \mid q_i] = q \wedge 0 \le e$.

In this case the theorem reduces to:

$$
\begin{array}{l}
\text{If} \quad (\underline{A}\ i \mid 0 \le i \mid c \mid [q \wedge 0 \le e \wedge e < I]\ p\ [r] \\
\qquad\qquad\qquad \vdash_{TC_0} \\
\qquad\qquad\qquad c \mid [q \wedge 0 \le e \wedge e = I]\ S\ [r] \\
\quad) \\
\text{then Tvalid1}(c \mid [q \wedge 0 \le e]\ p\ [r]\ )
\end{array}
$$

which we will write, in analogy to the partial correctness induction rule, as

$$\frac{c \mid [q \wedge 0 \leq e \wedge e < I]\ p\ [r] \ \vdash_{TC_0}\ c \mid [q \wedge 0 \leq e \wedge e = I]\ S\ [r]}{c \mid [q \wedge 0 \leq e]\ p\ [r]}$$

4.3.2.3. A note on the induction rules and their proofs

It may seem strange that the induction rules for partial and total correctness, which look so much alike, require rather different proofs. In this section we summarize the structure of these proofs, so as to clarify the differences. In fact, these differences were already present in the proof rules PR4 and TR4 for the DO-construct given in chapter 3, which were based on theorems 3.46.2 and 3.47.2, respectively. In theorems 4.33 and 4.11, however, the differences are much more pronounced. Let us reconsider the structure of their proofs.

The proof of theorem 4.33, the partial correctness case, is essentially of the following form:

> Let q_1 and q_2 be two conditions.
> Let Ψ be as in definition 4.26.2.
> From the premiss and the extended soundness notion it follows that for all ϕ: $(p \sqsubseteq \phi q) \Rightarrow (p \sqsubseteq \Psi(\phi)q)$.
> As the base step and admissibility are trivially satisfied, it follows by greatest fixed point induction that $p \sqsubseteq (\nu\Psi)q$.

The proof of theorem 4.41, the total correctness case, is essentially of the following form:

> Let $\langle q_i \rangle_{i=0}^{\infty}$ be a sequence of conditions
> Let r be a condition.
> Let Φ be as in definition 4.26.1, and let $\varphi = \mu\Phi$.
>
> premiss
>
> $\Rightarrow$ (for all i: $0 \leq i$: $(\bigsqcup_{0 \leq j < i} q_j \sqsubseteq \varphi r) \Rightarrow (q_i \sqsubseteq \Phi(\varphi)r)$)
>
> $\Rightarrow$ {fixed point property: $\varphi = \Phi(\varphi)$}
>
> (for all i: $0 \leq i$: $(\bigsqcup_{0 \leq j < i} q_j \sqsubseteq \varphi r) \Rightarrow (q_i \sqsubseteq \varphi r)$)
>
> {lemma 3.29}
>
> $\bigsqcup_{i=0}^{\infty} q_i \sqsubseteq r$.

Note that in the latter proof it is not necessary that φ is the least fixed point of Φ; only use has been made of the fixed point property $\varphi = \Phi(\varphi)$.

4.4. Recursive procedures with parameters

4.4.0. Introduction

In this section we will study programs of the form

$$p_1 = A_1, \ldots, p_n = A_n \mid B ,$$

where the p_i are identifiers, the A_i are abstractions, and B is a block. The constructs $p_i = A_i$ are to be considered as procedure declarations. Within the abstractions A_i and the block B statements may occur of the form p_j(EL;VL), where EL is a list of expressions and VL is a list of variables with types and lengths that match those of the abstraction A_j corresponding to p_j. The semantics will be defined in such a way that the statement p_j(EL;VL) is equivalent to the statement A_j(EL;VL), even if p_j has been defined recursively.

The treatment of this subject is essentially a combination of the treatments of parameterization and recursion in sections 4.2 and 4.3, respectively. In fact (and intentionally) there is so much correspondence that in many places we have taken the liberty to replace (parts of) definitions and proofs by an appeal to the similarity to their counterparts in those sections. In section 4.4.1 we deal with the syntactic aspects, in 4.4.2 with the semantics and in 4.4.3 with the proof rules.

4.4.1. Syntax

Apart from some additional production rules the main syntactic extension is the introduction of a new attribute domain *Penv* to establish the correspondence between declaration and use of procedure identifiers. The role of *Penv*-attributes is comparable to that of *Env*-attributes. It would have been possible to combine the two, but for the sake of clarity we have refrained from doing so. Below we give the extensions, followed by some informal explanation.

Domains

Penv

Operations on *Penv*

$[\cdot,\cdot,\cdot]_P$: *Name* * *Types* * *Types* → *Penv*

$\cdot$ $\backslash\!\!\textcircled{P}\!\!/$ $\cdot$: *Penv* * *Penv* → *Penv*

$\#_P(\cdot,\cdot)$: *Name* * *Penv* → *Int*

$(\cdot,\cdot,\cdot)$ $\underline{in}_P$ $\cdot$: *Name* * *Types* * *Types* * *Penv* → *Bool*

$\#_P(n_1,[n_2,ts_1,ts_2]_P) = \underline{if}\ n_1 = n_2 \rightarrow 1 \ [] \ n_1 \neq n_2 \rightarrow 0\ \underline{fi}$

$\#_P(n,pe_1 \backslash\!\!\textcircled{P}\!\!/ pe_2) = \#_P(n,pe_1) + \#_P(n,pe_2)$

$(n_1,ts_1,ts_2)\ \underline{in}_P\ [n_2,ts_3,ts_4]_P = (n_1 = n_2 \wedge ts_1 = ts_3 \wedge ts_2 = ts_4)$

$(n,ts_1,ts_2)\ \underline{in}_P\ pe_1 \backslash\!\!\textcircled{P}\!\!/ pe_2 = ((n,ts_1,ts_2)\ \underline{in}_P\ pe_1) \vee ((n,ts_1,ts_2)\ \underline{in}_P\ pe_2)$

Nonterminals

Procdecs <*Penv*,*Penv*>, Abstr <*Penv*,*Types*,*Types*>, Block <*Penv*,*Env*>, Stat <*Penv*,*Env*>, Gcs <*Penv*,*Env*>.
{see note below}

Grammar rules

Prog ::= Procdecs <pe_1,pe_2> | Block <pe_1,e> ■

$pe_1 = pe_2$
($\underline{A}$ n: *Name* | $\#_P(n,pe_2) \leq 1$)
e = Empty

Procdecs <pe_0,pe_1> ::= Procdecs <pe_0,pe_2> , Prodecs <pe_0,pe_3>

$pe_1 = pe_2 \backslash\!\!\textcircled{P}\!\!/ pe_3$

Procdecs <pe_0,pe_1> ::= Id <n> = Abstr <pe_0,ts_1,ts_2> ■

$pe_1 = [n,ts_1,ts_2]_P$

Abstr <pe,ts_1,ts_2> ::= ($\underline{con}$ Pdecs <d_1,ts_1>
; $\underline{res}$ Pdecs <d_2,ts_2>
| Stat <pe,e>
) ■

($\underline{A}$ n: *Name* | $\#_D(n,d_1 \backslash\!\!\textcircled{D}\!\!/ d_2) + \#_P(n,pe) \leq 1$)
e = Ext(Empty,$d_1 \backslash\!\!\textcircled{D}\!\!/ d_2$)

Stat <pe,e> ::= Id <n> (Exprs <e,ts_1> ;
Vars <e,ns,ts_2>) ■
(A n: *Name* | $\#_N$ (n,ns) ≤ 1)
(n,ts_1,ts_2) in_P pe

Explanation

The *Penv*-attributes establish the correspondence between procedure names and the parameter types of the corresponding abstractions. With the nonterminal Procdecs two such attributes are associated. The first one records information about all procedures declared in the procedure declaration part of a program; it is used to describe the legitimacy of procedure applications. The second one contains information concerning the internal procedure declarations. At the outermost level these attributes must be equal. The rules for formal-actual parameter correspondence are the same as in section 4.2.1. Abstractions have no access to global variables. Names of formal parameters must differ from procedure names (see also note 1 below).
□

Note 1

A *Penv*-attribute has also been associated with the nonterminals Block, Stat, and Gcs. Strictly speaking we should give new versions of the grammar rules for these nonterminals, However, as in most of these rules the *Penv*-attribute is merely "passed on", as e.g. in

Stat <pe,e> ::= Stat <pe,e>; Stat <pe,e> ■

we will not list them anew. The only exception is the rule for Block, which obtains the additional rule condition that variable names must differ from procedure names:

Block <pe,e_0> ::= |[var Decs <d> | Stat <pe,e_1>]| ■
(A n: *Name* | $\#_D$ (n,d) + $\#_P$ (n,pe) ≤ 1)
e_1 = Ext(e_0,d)

□

Note

As in section 4.2.1, an additional context condition is formulated by means of the functions USE, ASSN, and INIT. This part is almost identical to definitions 4.13 and 4.14 and the context condition following them, and therefore it is not repeated here.

□

4.4.2. Semantics

In this section we define the semantics of programs of the form $p_1 = A_1,\ldots,p_k = A_k \mid B$. Basically this definition has the same structure as that in section 4.3.1, the main difference being that with each name p_i a parameterized condition transformer rather than a condition transformer has to be associated. Similarly to section 4.3.1, these associations are established by means of an argument δ added to wp, wlp, pwp, and pwlp. Omitting environments for a while, the central clauses of the new definitions are:

$$wp'(p(E;v))\delta = pwp'(p)\delta(E,v)$$

and $pwp'(p)\delta = \delta(p)$.

The δ corresponding to a procedure declaration part is determined by means of fixed point methods.

An essential complication is that procedure applications may occur in different contexts. We recall definition 4.15, which defines P_e as the set of parameterized condition transformers corresponding to an environment $e \in \underline{Env}$. It follows that δ has to be parameterized with regard to the environment e of the procedure application, so as to obtain the appropriate element of P_e. Hence $\delta \in \underline{Name} \to \underline{Env} \to \mathrm{U}_{e\in\underline{Env}}\ P_e$, in such a way that for all procedure names p and $e \in \underline{Env}$: $\delta(p)(e) \in P_e$.

Apart from the points just mentioned, the definitions given in this section are very similar to those in sections 4.2.2 and 4.3.1, so we abstain from further clarification. The definitions follow.

Definition 4.42 {X}

$$X = \{\varphi \in \underline{Env} \to \mathop{\mathrm{U}}_{e\in\underline{Env}} P_e \mid (\underline{A}\ e \in \underline{Env} \mid \varphi(e) \in P_e)\} .$$

□

Definition 4.43 {$\sqsubseteq_X$}
For all $\varphi_1, \varphi_2 \in X$:

$$\varphi_1 \sqsubseteq_X \varphi_2 \quad \text{fif} \quad (\underline{A}\ e \in \underline{Env} \mid \varphi_1(e) \sqsubseteq_{P_e} \varphi_2(e)) .$$

□

Theorem 4.44
$(X, \sqsubseteq_X)$ is a ccl.
□

Proof
That $\sqsubseteq_X$ is a partial order follows immediately from definition 4.43, theorem 4.15 and theorem 3.10.2.
Let Y be a countable subset of X.

$$\sqcup Y = (\lambda e \in \underline{Env} \mid \bigsqcup_{\varphi \in Y} \lfloor P_e \rfloor\ \varphi(e)) .$$

$$\sqcap Y = (\lambda e \in \underline{Env} \mid \bigsqcap_{\varphi \in Y} \lceil P_e \rceil\ \varphi(e)) .$$

□

Definition 4.45 {Δ}

$$\Delta = \underline{Name} \to X .$$

Definition 4.46 {wp'}
For all $e \in \underline{Env}$ the function $wp'_e \in L(\text{Stat } \langle Penv, e\rangle) \to \Delta \to T_e$ is defined by:

1-6. Similar to definition 4.23.1-6.

7. $wp'_e(A(E;v))\delta = pwp'_e(A)(\delta)(E,v)$.

8. $wp'_e(p(E;v))\delta = pwp'_e(p)(\delta)(E,v)$.

□

Definition 4.47 {wlp'}
For all $e \in \underline{Env}$ the function $wlp'_e \in L(\text{Stat } \langle Penv, e\rangle) \to \Delta \to T_e$ is defined by:

1-6. Similar to definition 4.24.1-6.

7. $wlp'_e(A(E;v))\delta = pwlp'_e(A)(\delta)(E,v)$.

8. $wlp'_e(p(E;v))\delta = pwlp'_e(p)(\delta)(E,v)$.

□

Definition 4.48 {pwp', pwlp'}

For all e ∈ *Env* the functions pwp'_e and $pwlp'_e \in L(\text{Id } \langle Name \rangle) \cup L(\text{Abstr } \langle Penv, Types, Types \rangle) \to \Delta \to P_e$ are defined as follows:

Let (con x: t_1; res y: t_2 | S) ∈ $L(\text{Abstr } \langle Penv, Types, Types \rangle)$,
E, V, x', y', e', S' as in definition 4.17.

1.1. $pwp'_e((\text{con } x: t_1; \text{ res } y: t_2 \mid S))\delta =$

$(\lambda E \in E, v \in V \mid (x' \leftarrow E) \circ wp'_{e'}(S')\delta \circ (v \leftarrow y'))$.

1.2. $pwp'_e(p)\delta = \delta(p)(e)$.

2.1. $pwlp'_e((\text{con } x: t_1; \text{ res } y: t_2 \mid S))\delta =$

$(\lambda E \in E, v \in V \mid (x' \leftarrow E) \circ wlp'_{e'}(S')\delta \circ (v \leftarrow y'))$.

2.2. $pwlp'_e(p)\delta = \delta(p)(e)$.

□

Note

Continuity of the functions defined above can be proven similarly to theorem 4.25. Details omitted.

□

Definition 4.49 {wp and wlp for programs}

1. $wp(p_1 = A_1, \ldots, p_k = A_k \mid B) = wp'_{e_0}(B)\delta$,

where e_0 = Empty,
δ is the function $\{(p_1,\varphi_1), \ldots, (p_k,\varphi_k)\}$,
$(\varphi_1, \ldots, \varphi_k) = \mu(\Phi_1, \ldots, \Phi_k)$,
and, for i: $1 \le i \le k$:

$\Phi_i = (\lambda\varphi'_1, \ldots, \varphi'_k \in X \mid (\lambda e \in Env \mid pwp'_e(A_i)\{(p_1,\varphi'_1), \ldots, (p_k,\varphi'_k)\}))$.

2. $wlp(p_1 = A_1, \ldots, p_k = A_k \mid B) = wlp'_{e_0}(B)\delta$,

where e_0 = Empty,

δ is the function $\{(p_1,\phi_1),\ldots,(p_k,\phi_k)\}$,
$(\phi_1,\ldots,\phi_k) = \nu(\Psi_1,\ldots,\Psi_k)$,
and, for i: $1 \le i \le k$:

$\Psi_i = (\lambda\phi'_1,\ldots,\phi'_k \in X \mid (\lambda e \in \underline{Env} \mid pwlp'_e(A_i)\{(p_1,\phi'_1),\ldots,(p_k,\phi'_k)\}))$.

□

Note

Continuity of the functions Φ_i and Ψ_i can be proven similarly to theorem 4.27. Details omitted.

□

The following analogue of theorem 4.28 states the equivalence of the statements $p_i(E;v)$ and $A_i(E;v)$ when $p_i = A_i$ is a procedure declaration.

Theorem 4.50

Let $p_1 = A_1,\ldots,p_k = A_k \mid B$ be a program,
$e \in \underline{Env}$,
i: $1 \le i \le k$,
E, v such that $A_i(E;v) \in L(\text{Stat } \langle Penv,e\rangle)$.

1. Let δ be as in definition 4.49.1.

   $wp'_e(p_i(E;v))\delta = wp'_e(A_i(E;v))\delta$.

2. Let δ be as in definition 4.49.2.

   $wlp'_e(p_i(E;v))\delta = wlp'_e(A_i(E;v))\delta$.

□

Proof

We only prove 1; the proof of 2 is similar.

$wp'_e(p_i(E;v))\delta$

= {definition 4.46.8}

$pwp'_e(p_i)(\delta)(E,v)$

= {definition 4.48.1.2}

$\delta(p_i)(e)(E,v)$

= {definition 4.49.1, δ}

$\varphi_i(e)(E,v)$

= {fixed point property}

$\Phi_i(\varphi_1,\ldots,\varphi_k)(e)(E,v)$

= {definition 4.49.1, Φ_i}

$pwp'_e(A_i)(\delta)(E,v)$

= {definition 4.46.7}

$wp'_e(A_i(E;v))\delta$.

□

4.4.3. Proof rules

4.4.3.1. Proof rules for partial correctness

In this section we discuss partial correctness proof rules for programs of the form $p_1 = A_1,\ldots,p_k = A_k \mid B$. The structure of this section resembles that of section 4.3.2.1. First we define validity and soundness relatively to a set of procedure declarations, and we prove soundness of a rule to be used with nonrecursive procedures. Thereafter we consider an induction rule for use with recursive procedures. As in section 4.3.2.1, the proof of the latter rule requires definitions of validity and soundness relatively to an arbitrary interpretation ϕ for procedure names.

Definition 4.51 {Pvalid3}
On "partial correctness formulae" × "procedure declarations" the predicate Pvalid3 is defined as:

1. Pvalid3(c | q, $(p_1=A_1,\ldots,p_k=A_k \mid B)$) = $\vdash_D$ c | q .

2. Pvalid3(c | $\{q_1\}$ S $\{q_2\}$, $(p_1=A_1,\ldots,p_k=A_k \mid B)$) =

$\vdash_D$ c | $q_1 \Rightarrow wlp'_e(S)(\delta)(q_2)$,

where c ∈ L(Cont <e>), q_1,q_2 ∈ L(Cond <e>), and δ is as in definition 4.49.2.

□

Definition 4.52 {Psound3}
On "partial correctness proof rules" × "procedure declarations" the predicate Psound3 is defined as:

$$\text{Psound3}\left(\frac{c_0 \mid f_0,\ldots,f_{n-1}}{c_1 \mid f_n}\ ,\ pd\right) =$$

$$(\underline{A}\ i \mid 0\leq i\leq n \mid \text{Pvalid3}(c_0 \mid f_i,\ pd)) \Rightarrow \text{Pvalid3}(c_1 \mid f_n,\ pd)\ .$$

□

It is easy to prove that the axioms and proof rules considered in sections 3.3, 4.1.4, and 4.2.3 are valid c.q. sound in terms of these extended notions as well. For easier reference we collect these axioms and rules under the name PC_1:

Definition 4.53 {PC_1}
The partial correctness logic PC_1 is defined by:

$$Ax_{PC_1} = Ax_{PC_0}\ ,$$

$$Pr_{PC_1} = Pr_{PC_0} \cup \{PR5,PR6,PR7,PR8,PR9\}\ .$$

□

Theorem 4.54

1. $(\underline{A}\ a \in Ax_{PC_1} \mid \text{Pvalid3}(a,\ (p_1=A_1,\ldots,p_k=A_k)))$.
2. $(\underline{A}\ r \in Pr_{PC_1} \mid \text{Psound3}(r,\ (p_1=A_1,\ldots,p_k=A_k)))$.

□

Proof
Similar to those of theorems 3.52, 4.12, and 4.20. Details omitted.
□

Below, for simplicity we restrict ourselves to the case k = 1. The results can easily be extended to programs with more procedure declarations. First, we present proof rules for use with nonrecursive procedures.

Definition 4.55 {PR10, PR11}

Let $A = (\underline{con}\ x\colon t_1;\ \underline{res}\ y\colon t_2 \mid S) \in L(\text{Abstr } \langle Penv, Types, Types\rangle)$.
Let E, v, c, C, q_1, q_2 be as in definition 4.19.

The proof rules PR10 and PR11 are defined by:

$$\text{PR10}\quad \frac{x\colon t_1,\ y\colon t_2 \mid \{q_1(x)\}\ S\ \{q_2(x,y)\}}{c \mid \{q_1(E)\}\ p(E;v)\ \{q_2(E,v)\}}$$

provided $v \notin USE(E)$.

$$\text{PR11.}\quad \frac{x\colon t_1,\ y\colon t_2 \mid \{q_1(x)\}\ S\ \{q_2(x,y)\}}{c \triangleright C\colon t_1 \mid \{E{=}C \wedge q_1(C)\}\ p(E;v)\ \{q_2(C,v)\}}$$

□

Theorem 4.56

1. Psound3(PR10, p = A) .
2. Psound3(PR11, p = A) .

□

Proof

We only consider 1; 2 is similar.

Assume:

Pvalid3(x: t_1, y: t_2 | $\{q_1(x)\}$ S $\{q_2(x,y)\}$, p = A) .

By a proof almost identical to that of theorem 4.20 we obtain:

Pvalid3(c | $\{q_1(E)\}$ A(E;v) $\{q_2(E,v)\}$, p = A)

provided $v \notin USE(E)$.

Hence, by definition 4.51 and theorem 4.50.2:

Pvalid3(c | $\{q_1(E)\}$ p(E;v) $\{q_2(E,v)\}$, p = A)

provided $v \notin USE(E)$.

□

Let us now turn to the induction rule for recursive procedures. The treatment of this rule is similar to that of the parameterless induction rule in section 4.3.2.1. First we define validity and soundness

with regard to an arbitrary interpretation ϕ for procedure names. By means of these notions we prove theorem 4.61, which is finally presented as inductive proof rule PR_{12}.

Definition 4.57 {Pvalid4}
On "partial correctness formulae" × X the predicate Pvalid4 is defined as:

1. $Pvalid4(c \mid q, \phi) = \vdash_D c \mid q$.

2. $Pvalid4(c \mid \{q_1\}\ S\ \{q_2\}, \phi) =$

$$\vdash_D c \mid q_1 \Rightarrow wlp'_e(S)\{(p,\phi)\}q_2 ,$$

where $c \in L(\text{Cont } \langle e \rangle)$, $q_1, q_2 \in L(\text{Cond } \langle e \rangle)$.

□

Definition 4.58 {Psound4}
On "partial correctness proof rules" × X the predicate Psound4 is defined as:

$$Psound4\left(\frac{c_0 \mid f_0,\ldots,f_{n-1}}{c_1 \mid f_n}, \phi\right) =$$

$$(\underline{A}\ i \mid 0 \le i \le n \mid Pvalid4(c_0 \mid f_i, \phi)) \Rightarrow Pvalid4(c_1 \mid f_n, \phi) .$$

□

Theorem 4.60

1. $(\underline{A}\ a \in Ax_{PC_1}, \phi \in X \mid Pvalid4(a,\phi))$.

2. $(\underline{A}\ r \in Pr_{PC_1}, \phi \in X \mid Psound4(r,\phi))$.

□

Proof

Similar to those of theorems 3.52, 4.12, and 4.20. Details omitted.
□

Theorem 4.61

Let $A = (\text{con } x\text{: } t_1\text{; res } y\text{: } t_2 \mid S) \in L(\text{Abstr } \langle \mathit{Penv}, \mathit{Types}, \mathit{Types} \rangle)$.

Let $e' = \text{Ext}(\text{Empty}, [x,t_1]_D \mathbin{\textcircled{D}} [y,t_2]_D)$,

$q_1(x), q_2(x,y) \in L(\text{Cond } \langle e' \rangle)$.

For i: $0 \leq i \leq n$:

let $e_i \in \mathit{Env}$,

$c_i \in L(\text{Cont } \langle e_i \rangle)$,

$E_i \in L(\text{Expr } \langle e_i, \mathit{Prio}, t_1 \rangle)$,

$v_i \in L(\text{Var } \langle e_i, \mathit{Name}, t_2 \rangle)$, $v_i \notin \text{USE}(E_i)$.

If

$$c_1 \mid \{q_1(E_1)\}\ p(E_1;v_1)\ \{q_2(E_1,v_1)\} ,$$
$$\ldots$$
$$c_n \mid \{q_n(E_n)\}\ p(E_n;v_n)\ \{q_2(E_n,v_n)\}$$

$$\vdash_{PC_1}$$

$$x\text{: } t_1,\ y\text{: } t_2 \mid \{q_1(x)\}\ S\ \{q_2(x,y)\}$$

then

$$\text{Pvalid3}(c_0 \mid \{q_1(E_0)\}\ p(E_0;v_0)\ \{q_2(E_0,v_0)\},\ p = A) .$$

□

Proof

The proof is by greatest fixed point induction. In the proof we use a kind of "phase shift" in that we prove validity of correctness formulae involving S rather than p(E;v) for arbitrary E and v. This phase shift leads to shorter formulae. The property we want to prove is:

$$\text{Pvalid4}(x\text{: } t_1;\ y\text{: } t_2 \mid \{q_1(x)\}\ S\ \{q_2(x,y)\},\ \nu\Psi) ,$$

where Ψ is as in definition 4.49.2.

base step

theorem 4.44

$\Rightarrow {}^{\tau}X = (\lambda e \in \mathit{Env} \mid {}^{\tau}P_e)$

$\Rightarrow$ {definitions 4.47.8, 4.48.2.2}

$(\underline{A}\ i \mid 1 \le i \le n \mid \mathrm{wlp}'_{e_i}(p(E_i;v_i))\{(p,\tau_X)\} = (\lambda q \mid \mathrm{true}))$

$\Rightarrow$ {definition 4.57}

$(\underline{A}\ i \mid 1 \le i \le n \mid \mathrm{Pvalid4}(c_i \mid \{q_1(E_i)\}\ p(E_i;v_i)\ \{q_2(E_i,v_i)\},\ \tau_X))$

$\Rightarrow$ {premiss, theorem 4.60}

$\mathrm{Pvalid4}(x{:}\ t_1;\ t{:}\ t_2 \mid \{q_1(x)\}\ S\ \{q_2(x,y)\},\ \tau_X)$.

induction step

Let $\psi \in X$.

$\mathrm{Pvalid4}(x{:}\ t_1;\ y{:}\ t_2 \mid \{q_1(x)\}\ S\ \{q_2(x,y)\},\ \psi)$

$\Rightarrow$ {theorem 4.60.2, hence Psound4 (PR8)}

$(\underline{A}\ i \mid 1 \le i \le n \mid \mathrm{Pvalid4}(c_i \mid \{q_1(E_i)\}\ A(E_i;v_i)\ \{q_2(E_i,v_i)\},\ \psi))$

$=$ {definition 4.57}

$(\underline{A}\ i \mid 1 \le i \le n \mid$

$\quad \vdash_D c_i \mid q_1(E_i) \Rightarrow \mathrm{pwlp}'_{e_i}(A_i)\{(p,\psi)\}(E_i,v_i)q_2(E_i,v_i)$

$)$

$=$ {definition 4.49.2, Ψ}

$(\underline{A}\ i \mid 1 \le i \le n \mid \vdash_D c_i \mid q_1(E_i) \Rightarrow \Psi(\psi)(e)(E_i,v_i)q_2(E_i,v_i))$

$=$ {definitions 4.47.8, 4.48.2.2}

$(\underline{A}\ i \mid 1 \le i \le n \mid$

$\quad \vdash_D c_i \mid q(E_i) \Rightarrow \mathrm{wlp}'_{e_i}(p(E_i;v_i))\{(p,\Psi(\psi))\}q_2(E_i,v_i)$

$)$

$=$ {definition 4.57}

$(\underline{A}\ i \mid 1 \le i \le n \mid \mathrm{Pvalid4}(c_i \mid \{q_1(E_i)\}\ p(E_i;v_i)\ \{q_2(E_i,v_i)\},\ \Psi(\psi)))$

$\Rightarrow$ {premiss, theorem 4.60}

$\mathrm{Pvalid4}(x{:}\ t_1;\ y{:}\ t_2 \mid \{q_1(x)\}\ S\ \{q_2(x,y)\},\ \Psi(\psi))$.

As admissibility is trivial, it follows by greatest fixed point induction that

Pvalid4(x: t_1; y: t_2 | $\{q_1(x)\}$ S $\{q_2(x,y)\}$, $\nu\Psi$)

= {definitions 4.57.2, 4.51.2}

Pvalid3(x: t_1; y: t_2 | $\{q_1(x)\}$ S $\{q_2(x,y)\}$, p = A)

⇒ {theorem 4.56.1}

Pvalid3(c_0 | $\{q_1(E_0)\}$ $p(E_0;v_0)$ $\{q_2(E_0,v_0)\}$, p = A)

□

Finally, in analogy with the parameterless case in theorem 4.33, we formulate theorem 4.61 as an inductive proof rule:

PR12.

$$\frac{\begin{array}{l} c_1 \mid \{q_1(E_1)\}\ p(E_1;v_1)\ \{q_2(E_1,v_1)\} \\ \cdots \\ c_n \mid \{q_1(E_n)\}\ p(E_n;v_n)\ \{q_2(E_n,v_n)\} \\ \vdash_{PC_1} \\ x\colon t_1,\ y\colon t_2 \mid \{q_1(x)\}\ S\ \{q_2(x,y)\} \end{array}}{c_0 \mid \{q_1(E_0)\}\ p(E_0;v_0)\ \{q_2(E_0,v_0)\}}$$

4.4.3.2. Proof rules for total correctness

In this section we discuss total correctness proof rules for programs of the form $p_1 = A_1,\ldots,p_k = A_k \mid B$. As this section has much in common with sections 4.3.2.2 and 4.4.3.1, we will shorten the presentation somewhat. The main theorem of this section is theorem 4.68, the induction rule for recursive procedures, also formulated as proof rule TR12. The structure of the proof is essentially the same as that of theorem 4.41, the parameterless induction rule.

Definition 4.62 {Tvalid2}
Similar to definitions 4.37 and 4.51.
□

Definition 4.63 {Tsound2}
Similar to definitions 4.38 and 4.52.
□

Definition 4.64 {TC_1}

The total correctness logic TC_1 is defined by

$$Ax_{TC_1} = Ax_{TC_0} ,$$

$$Pr_{TC_1} = Pr_{TC_0} \cup \{TR5, TR6, TR7, TR8, TR9\} .$$

□

Theorem 4.65

1. $(\underline{A}\ a \in Ax_{TC_1} \mid Tvalid2(a, (p_1=A_1, \ldots, p_k=A_k))$.

2. $(\underline{A}\ r \in Pr_{TC_1} \mid Tsound2(r, (p_1=A_1, \ldots, p_k=A_k))$.

□

Proof

Similar to those of theorems 3.54, 4.12, and 4.20. Details omitted.

□

Definition 4.66 {TR10, TR11}

Let $A = (\underline{con}\ x\colon t_1;\ \underline{res}\ y\colon t_2 \mid S) \in L(\text{Abstr } \langle Penv, Types, Types\rangle)$.
Let E, v, c, C, q_1, q_2 be as in definition 4.19.

The proof rules TR10 and TR11 are defined by:

$$\text{TR10.}\quad \frac{x\colon t_1,\ y\colon t_2 \mid [q_1(x)]\ S\ [q_2(x,y)]}{c \mid [q_1(E)]\ p(E;v)\ [q_2(E,v)]}$$

provided $v \notin USE(E)$.

$$\text{TR11.}\quad \frac{x\colon t_1,\ y\colon t_2 \mid [q_1(x)]\ S\ [q_2(x,y)]}{c \rhd C\colon t_1 \mid [E=C \wedge q_1(C)]\ p(E;v)\ [q_2(C,v)]}$$

□

Theorem 4.67

1. Tsound2 (TR10, p = A) .

2. Tsound2 (TR11, p = A) .

□

Proof

Similar to that of theorem 4.56. Details omitted.

□

Theorem 4.68

Let $A = (\underline{con}\ x\colon t_1;\ \underline{res}\ y\colon t_2 \mid S) \in L(\text{Abstr } \langle Penv, Types, Types\rangle)$.

For i: $0 \le i \le n$: let e_i, c_i, E_i, v_i be as in theorem 4.61.

Let $\langle q_j(x)\rangle_{j=0}^{\infty}$ be a sequence of conditions in
$L(\text{Cond } \langle \text{Ext}(\text{Empty}, [x,t_1]_D)\rangle)$.

Let $r(x,y) \in L(\text{Cond } \langle \text{Ext}(\text{Empty}, [x,t_1]_D \,\mathbb{D}\, [y,t_2]_D)\rangle)$.

If

$$(\underline{A}\ k \mid 0\le k \mid$$
$$c_1 \mid [[\vee j \mid 0\le j<k \mid q_j(E_1)]]\ p(E_1;v_1)\ [r(E_1,v_1)]$$
$$\cdots$$
$$c_n \mid [[\vee j \mid 0\le j<k \mid q_j(E_n)]]\ p(E_n;v_n)\ [r(E_n,v_n)]$$
$$\vdash_{TC_1}$$
$$x\colon t_1,\ y\colon t_2 \mid [q_k(x)]\ S\ [r(x,y)]$$
$$)$$

then

$$\text{Tvalid2}(c_0 \mid [[\vee k \mid 0\le k \mid q_k(E_0)]]\ p(E_0;v_0)\ [r(E_0,v_0)],\ p = A)\ .$$

□

Proof

Let k: $k \ge 0$.

$$\text{Tvalid2}(x\colon t_1,\ y\colon t_2 \mid [[\vee j \mid 0\le j<k \mid q_j(x)]]\ S\ [r(x,y)],\ p = A)$$
$$\Rightarrow \{\text{theorem 4.67.1}\}$$
$$(\underline{A}\ i \mid 1\le i\le n \mid$$
$$\text{Tvalid2}(c_i \mid [[\vee j \mid 0\le j<k \mid q_j(E_i)]]\ p(E_i;v_i)\ [r(E_i,v_i)],\ p = A)$$
$$)$$
$$\Rightarrow \{\text{premiss of theorem, theorem 4.65}\}$$
$$\text{Tvalid2}(x\colon t_1,\ y\colon t_2 \mid [q_k(x)]\ S\ [r(x,y)],\ p = A)\ .$$

By definitions 4.62, 3.33 and lemma 3.29 it follows that

$$\text{Tvalid2}(x\colon t_1,\ y\colon t_2 \mid [[\vee k \mid 0\le k \mid q_k(x)]]\ S\ [r(x,y)],\ p = A)\ ,$$

hence, by theorem 4.67.1,

$$\mathrm{Tvalid2}(c_0 \mid [[\bigvee k \mid 0 \le k \mid q_k(E_0)]]\ p(E_0;v_0)\ [r(E_0,v_0)],\ p = A)\ .$$

□

Just as in section 4.3.2.2, if we choose $q_k(x) = (q(x) \wedge h(x) = K)$, where K is a symbol sequence representing k and h(x) is an integer expression in terms of x, we obtain that

$$[\bigvee j \mid 0 \le j < k \mid q_j(E_i)] = q(E_i) \wedge 0 \le h(E_i) < K$$

and $[\bigvee k \mid 0 \le k \mid q_k(E_0)] = q(E_0) \wedge 0 \le h(E_0)$,

as a result of which the theorem can be formulated as the following inductive proof rule:

TR12.

$$\frac{\begin{array}{l} c_1 \mid [q(E_1) \wedge 0 \le h(E_1) < K]\ p(E_1;v_1)\ [r(E_1,v_1)] \\ \cdots \\ c_n \mid [q(E_n) \wedge 0 \le h(E_n) < K]\ p(E_n;v_n)\ [r(E_n,v_n)] \\ \vdash_{TC_1} \\ x\colon t_1,\ y\colon t_2 \mid [q(x) \wedge h(x) = K]\ S\ [r(x,y)] \end{array}}{c_0 \mid [q(E_0) \wedge 0 \le h(E_0)]\ p(E_0;v_0)\ [r(E_0,v_0)]\ .}$$

CHAPTER 5
SOME ASPECTS OF THE DEFINITION OF THE TARGET LANGUAGE

5.0. Introduction

In this chapter we will consider some aspects of the formal definition of the target language TL. TL has a rather conventional structure; it contains instructions for loading and storing values, arithmetical operations, jumps, conditional jumps, subroutine calls and returns. Usually the effect of these instructions is described operationally in terms of manipulations with some registers and an instruction pointer. For our purposes we would like to have at our disposal a condition transformer definition of TL however. Such a definition will allow us to derive a mapping from SL to TL from relations between condition transformers of SL- and TL-programs.

For simple load and store instructions and the like it is easy to construct condition transformers based on that of the assignment. The difficulties are in the definition of the sequencing instructions. It is this aspect that we would like to consider in the current chapter. We shall develop condition transformers and show their equivalence with an operational characterization. Rather than presenting two definitions and proving their equivalence however, we shall derive the operational description from the condition transformers. This derivation will proceed via some intermediate versions in which more and more operational aspects are introduced. This chapter therefore has the following structure.

In section 5.1 we present an informal description of TL instructions. In section 5.2 we develop version 1 of their condition transformer semantics, which bears some relation to that of parameterless procedures in section 4.3 and to the continuations of denotational semantics. In section 5.3 we derive from version 1 a second version, which is based upon the representation of a TL program as an array of instructions. In section 5.4 this version is further transformed to

version 3, which employs stacks of return addresses to describe the subroutine call and return mechanism. Finally, in section 5.5 we derive from version 3 an operational characterization by means of an interpreting program. This interpreter is constructed in such a way that it has the same condtion transformer as that of the TL program in version 3, and consequently as that in version 1.

In some derivation steps use is made of the fixed point property in order to establish some relation between two versions. Those derivations are given in such a way that it is clear how a full fixed point induction proof of the equivalence of the two versions should be given.

5.1. Informal description of TL

For the purposes of this chapter, TL may be thought of as consisting of two sets of variables and of a set of instructions. The variables of the first set, called the data variables, are used to represent the values operated upon by TL programs. For our discussion the properties of these variables are not very important. For simplicity, let us assume that the set consists of

M	the store, a linear array of values,
A and B	two general purpose registers,
Q	a condition register.

The second set of variables, the control variables, is mainly used for sequencing purposes. It consists of

P	the program store, a linear array of instructions,
ip	the instruction pointer,
rs	the stack of return points,
la	a function that maps labels to addresses.

A TL program is a construct of the form

$$l_0: s_0; \ \cdots \ ; l_{n-1}: s_{n-1}; l_n:$$

where the l_i are labels and the s_i are instruction sequences. We distinguish two sorts of instructions, regular and singular instructions. An instruction is regular if the instruction to be executed after its completion is its textual successor; an instruction is singular, if it explicitly alters the flow of control, as is the case with jumps, subroutine calls and returns.
Typically, the effect of regular instructions can be described by means of a few assignments to the data variables; e.g.:

LDA(a)	A := M(a)
STB(a)	M(a) := B
ADD(A,B)	A := A + B
GEQ	Q := A ≥ B .

The description of singular instructions is less easy. Their formal characterization is the main subject of this chapter. Here we content ourselves with a short description in words.

UJP(1)	jump unconditionally to label 1,
FJP(1)	if Q is false jump to label 1,
TJP(1)	if Q is true jump to label 1,
CSR(1)	record return point, jump to label 1,
RET	remove last return point r from record; jump to r.

We shall assume that programs are closed, i.e. that each label occurring in a singular instruction of a program l_0: s_0; ... ; l_{n-1}: s_{n-1}; l_n: is an element of $\{l_0,\ldots,l_n\}$.

5.2. Version 1: Condition transformer semantics of TL

In this section we shall develop condition transformer semantics for closed TL programs. As we are mainly interested in control flow aspects, we shall restrict ourselves to the singular instructions and to the regular instruction LDA(a), which serves as representative for all regular instructions. Throughout the remainder of this chapter (except for an example) we shall base our discussion on a given closed program

$$p = l_0: s_0; \ldots ; l_{n-1}: s_{n-1}; l_n:$$

consisting of these instructions only.

Definition 5.1 {Instruction}

$$\text{Instruction} = \{\text{LDA(a)},\text{UJP(l)},\text{FJP(l)},\text{TJP(l)},\text{CSR(l)},\text{RET}\} .$$

□

Definition 5.2 {Label}

$$\text{Label} = \{l_0,\ldots,l_n\} .$$

□

We assume as given a ccl $(C,\underline{\leq}_C)$ of conditions in terms of the data variables of TL.

Definition 5.3 $\{T,\underline{\leq}_T\}$

1. $T = C \rightarrow_{uc} C$.
2. $\underline{\leq}_T$ is the standard order on T.

□

Lemma 5.4

$(T,\underline{\leq}_T)$ is a uccl.

□

Proof

Immediately by theorem 3.23.2.

□

It is our intention to associate with program p a condition transformer, i.e. an element of T. To this end we would like to define a function τ that associates a condition transformer with each instruction sequence. In first approximation τ would be an element of Instruction $\rightarrow$ T. For regular instructions this would suffice; we could simply define

$$\tau[\![\text{LDA(a)}]\!] = (A \leftarrow M(a)) \quad \text{and} \quad \tau[\![s_1;s_2]\!] = \tau[\![s_1]\!] \circ \tau[\![s_2]\!]$$

(we use the brackets $[\![$ and $]\!]$ to enclose instruction sequences). This approach does not work for singular instructions; e.g. if in $\tau[\![s_1;s_2]\!]$ the last instruction of s_1 is UJP(l_i), $\tau[\![s_1]\!]$ should be composed with the condition transformer of the instruction sequence following l_i, not with $\tau[\![s_2]\!]$. The problem can be solved by applying the continuation technique of denotational semantics [Strachey]: $\tau[\![s_1]\!]$ is supplied with two parameters, a "label environment" le and a "normal continuation" nc. The parameter nc corresponds to the condition transformer of the instruction sequence textually following s_1; if s_1 ends with a regular instruction its condition transformer should be composed with nc. The label environment le $\in$ Label $\rightarrow$ T is comparable to the function δ of definitions 4.23 and 4.24. With each $l_i \in$ Label it associates the condition transformer of the instruction sequence following l_i. If s_1 contains singular instructions referring to a label l_j, the condition transformer of s_1 will depend upon le(l_j). Thus, in second approximation τ is defined by clauses like

$$\tau[\![\text{LDA(a)}]\!](\text{le},\text{nc}) = (A \leftarrow M(a)) \circ \text{nc} ,$$

$$\tau[\![\text{UJP(l)}]\!](\text{le},\text{nc}) = \text{le(l)} ,$$

$$\tau[\![s_1;s_2]\!](\text{le},\text{nc}) = \tau[\![s_1]\!](\text{le},\tau[\![s_2]\!](\text{le},\text{nc})) .$$

Still this form is insufficient, as it does not handle subroutine calls and returns. The condition transformer of the RET instruction should be composed neither with that of its textual successor, nor with le(l_i) for some label l_i, but with the condition transformer of the instruction sequence following the CSR instruction "last executed". This condition transformer, the "return continuation", should therefore be passed as an additional parameter rc to both τ and le. In this way the condition transformer of RET is simply rc, whereas the condition transformer of CSR(l) with a normal continuation nc is le(l,nc) in order to

establish that the condition transformer of a subsequent RET is that of the instruction sequence following the CSR(1) instruction. Thus we obtain, in third approximation, defining clauses like the following:

$$\tau[\![\text{LDA(a)}]\!](le,rc,nc) = (A \leftarrow M(a)) \circ nc ,$$

$$\tau[\![\text{UJP(1)}]\!](le,rc,nc) = le(1,rc) ,$$

$$\tau[\![\text{CSR(1)}]\!](le,rc,nc) = le(1,nc) ,$$

$$\tau[\![\text{RET}]\!](le,rc,nc) = rc ,$$

$$\tau[\![s_1;s_2]\!](le,rc,nc) = \tau[\![s_1]\!](le,rc,\tau[\![s_2]\!](le,rc,nc)) .$$

In principle a definition of this kind could do the job, but for future applications it will be more convenient to decompose τ into two functions ρ and σ, such that for single instructions i:

$$\tau[\![i]\!](le,rc,nc) = \rho[\![i]\!] \circ \sigma[\![i]\!](le,rc,nc) ,$$

where ρ describes the way the data variables are affected by i, and σ takes care of the sequencing. This decomposition gives rise to the fourth and final approximation presented in definitions 5.8 and 5.9 below. As preparation for these definitions we first define some abbreviating functions:

Definition 5.5

The functions $I,F \in T$, $CF \in C \times T \times T \to T$ are defined by

1. $I = (\lambda q \in C \mid q)$
2. $F = (\lambda q \in C \mid \text{false})$
3. $CF = (\lambda q_1 \in C;\ f_1,f_2 \in T \mid (\lambda q_2 \in C \mid q_1 \wedge f_1(q_2) \vee \neg q_1 \wedge f_2(q_2)))$.

□

The function CF will be used in the definition of the conditional jumps. Note that $CF(q,f_1,f_2) = CF(\neg q,f_2,f_1)$.

The function le ("label environment") is an element of the set Labenv defined below. In principle Labenv is the set Label $\times$ T $\to$ T, but in order to ensure continuity of some other functions to be defined later on, we restrict Labenv to functions that are continuous in their second argument.

Definition 5.6 {Labenv}

1. Labenv = {le ∈ Label × T → T

| for each l ∈ Label, each ascending chain $\langle t_i \rangle_{i=0}^{\infty}$ in T:

$le(l, \bigsqcup_{i=0}^{\infty} t_i) = \bigsqcup_{i=0}^{\infty} le(l, t_i)$

}

2. $\sqsubseteq_{LE}$ is the standard order on Labenv.

□

Lemma 5.7

(Labenv, $\sqsubseteq_{LE}$) is a uccl.

□

Proof

Similar to theorem 3.23.2.

□

Definition 5.8 {ρ, σ}

ρ ∈ Instruction → T .

σ ∈ Instruction → Labenv × T × T → T .

For all le ∈ Labenv, rc,nc ∈ T:

1.1. ρ⟦LDA(a)⟧ = (A ← M(a)) .

1.2. σ⟦LDA(a)⟧(le,rc,nc) = nc .

2.1. ρ⟦UJP(l)⟧ = I .

2.2. σ⟦UJP(l)⟧(le,rc,nc) = le(l,rc) .

3.1. ρ⟦FJP(l)⟧ = I .

3.2. σ⟦FJP(l)⟧(le,rc,nc) = CF(Q,nc,le(l,rc)) .

4.1. ρ⟦TJP(l)⟧ = I .

4.2. σ⟦TJP(l)⟧(le,rc,nc) = CF(¬Q,nc,le(l,rc)) .

5.1. ρ⟦CSR(l)⟧ = I .

5.2. σ⟦CSR(l)⟧(le,rc,nc) = le(l,nc) .

6.1. $\rho[\![\text{RET}]\!] = I$.

6.2. $\sigma[\![\text{RET}]\!](le,rc,nc) = rc$.

□

Definition 5.9 $\{\tau\}$

$$\tau \in \text{Instruction}^* \to \text{Labenv} \times T \times T \to T .$$

For all $le \in \text{Labenv}$, $rc,nc \in T$, $i \in \text{Instruction}$, $s_1,s_2 \in \text{Instruction}^*$:

1. $\tau[\![s_1;s_2]\!](le,rc,nc) = \tau[\![s_1]\!](le,rc,\tau[\![s_2]\!](le,rc,nc))$.

2. $\tau[\![i]\!](le,rc,nc) = \rho[\![i]\!] \circ \sigma[\![i]\!](le,rc,nc)$.

□

The following lemma states associativity of τ with regard to ";".

Lemma 5.10

For all $s_1,s_2,s_3 \in \text{Instruction}^*$, $le \in \text{Labenv}$, $rc,nc \in T$:

$$\tau[\![(s_1;s_2);s_3]\!](le,rc,nc) = \tau[\![s_1;(s_2;s_3)]\!](le,rc,nc) .$$

□

Proof

$\tau[\![(s_1;s_2);s_3]\!](le,rc,nc)$

= {definition 5.9.1}

$\tau[\![s_1;s_2]\!](le,rc,\tau[\![s_3]\!](le,rc,nc))$

= {definition 5.9.1}

$\tau[\![s_1]\!](le,rc,\tau[\![s_2]\!](le,rc,\tau[\![s_3]\!](le,rc,nc)))$

= {definition 5.9.1}

$\tau[\![s_1]\!](le,rc,\tau[\![s_2;s_3]\!](le,rc,nc))$

= {definition 5.9.1}

$\tau[\![s_1;(s_2;s_3)]\!](le,rc,nc)$.

□

Theorem 5.11

1. For all i ∈ Instruction : σ⟦i⟧ ∈ Labenv × T × T →$_{uc}$ T .

2. For all s ∈ Instruction*: τ⟦s⟧ ∈ Labenv × T × T →$_{uc}$ T .

□

Proof

We only prove 1 for the case i :: UJP(1). The other proofs are similar.

Let $\langle le_i \rangle_{i=0}^{\infty}$, $\langle rc_j \rangle_{j=0}^{\infty}$ and $\langle nc_k \rangle_{k=0}^{\infty}$ be ascending chains in Labenv, T and T, respectively.

$$\sigma⟦\text{UJP}(1)⟧\,(\bigsqcup_{i=0}^{\infty} le_i,\ \bigsqcup_{j=0}^{\infty} rc_j,\ \bigsqcup_{k=0}^{\infty} nc_k)$$

= {definition 5.8.2.2}

$$\bigsqcup_{i=0}^{\infty} le_i(1, \bigsqcup_{j=0}^{\infty} rc_j)$$

= {$le_i \in$ Labenv, definition 5.6}

$$\bigsqcup_{i=0}^{\infty} \bigsqcup_{j=0}^{\infty} le_i(1, rc_j)$$

= {lemma 3.20.1}

$$\bigsqcup_{i=0}^{\infty} le_i(1, rc_i)$$

= {definition 5.8.2.2}

$$\bigsqcup_{i=0}^{\infty} \sigma⟦\text{UJP}(1)⟧\,(le_i, rc_i, nc_i) \ .$$

□

Finally we define the function π which yields the condition transformer of an entire program l_0: s_0; l_1: s_1; ...; l_n: expressed in those of its constituent instruction sequences s_i. Similarly to δ in definition 4.26, the label environment le of an entire program is defined as least fixed point of a function F ∈ Labenv → Labenv. The condition transformer of the program is that of the instruction sequence following label l_0, i.e. $le(l_0, rc)$ for some suitable rc. We choose rc = F to ensure that a program aborts if it attempts to execute more RET than CSR instructions.

Definition 5.12 {π}

$$\pi \in \text{Program} \to T \ .$$

$$\pi⟦\, l_0\text{: } s_0\text{; } \cdots \text{; } l_{n-1}\text{: } s_{n-1}\text{; } l_n\text{: }⟧ = le(l_0, F) \ ,$$

where

$$le = \mu F ,$$

and $F \in Labenv \to Labenv$ is given by

$$F = (\lambda le' \in Labenv \mid$$
$$(\lambda l \in Label,\ rc \in T \mid [l = l_0 \to \tau[\![s_0]\!](le', rc, le'(l_1, rc)) ,$$
$$\ldots$$
$$l = l_{n-1} \to \tau[\![s_{n-1}]\!](le', rc, le'(l_n, rc)) ,$$
$$l = l_n \to I$$
$$]$$
$$)$$
$$)$$

□

In order for this definition to be well-formed, it is necessary that F is continuous. This is ensured by the following theorem.

Theorem 5.13

$$F \in (Labenv \to_{uc} Labenv) .$$

□

Proof

Immediately by theorem 5.11 and the "pointwise" lub definition in Labenv.

□

Example

Consider the programs x, y and z below, which correspond to common translations of the SL programs

$$p = \underline{\text{if}}\ B \to S_1;\ p \ [\!]\ \neg B \to S_2\ \underline{\text{fi}} \mid p ,$$

or $\underline{\text{do}}\ B \to S_1\ \underline{\text{od}};\ S_2$.

Let b, s_1 and s_2 be regular instructions.

```
x = lx0: UJP(lx4);
   lx1: b;
        FJP(lx2);
        s1;
        CSR(lx1);
        UJP(lx3);
   lx2: s2;
   lx3: RET;
   lx4: CSR(lx1);
   lx5:

y = ly0: b;
        FJP(ly1);
        s1;
        UJP(ly0);
   ly1: s2;
   ly2:

z = lz0: UJP(lz2);
   lz1: s1;
   lz2: b;
        TJP(lz1);
        s2;
   lz3:
```

By definition 5.12 and the fixed point property we obtain, after simplification by means of definitions 5.8 and 5.9:

$$
\begin{aligned}
lex = (&\lambda l, rc \mid \\
&[l = lx_0 \to lex(lx_4, rc) \\
&\ l = lx_1 \to \rho[b] \circ CF(Q, \rho[s_1] \circ lex(lx_1, lex(lx_3, rc)), \\
&\qquad\qquad lex(lx_2, rc)) , \\
&\ l = lx_2 \to \rho[s_2] \circ lex(lx_3, rc) , \\
&\ l = lx_3 \to rc , \\
&\ l = lx_4 \to lex(lx_1, lex(lx_5, rc)) , \\
&\ l = lx_5 \to I \\
&] \\
)&
\end{aligned}
$$

$$
\begin{aligned}
ley = (&\lambda l, rc \mid \\
&[l = ly_0 \to \rho[b] \circ CF(Q, \rho[s_1] \circ lex(ly_0, rc), lex(ly_1, rc)) , \\
&\ l = ly_1 \to \rho[s_2] \circ ley(ly_2, rc) , \\
&\ l = ly_2 \to I \\
&] \\
)&
\end{aligned}
$$

$$
\begin{aligned}
lez = (&\lambda l, rc \mid \\
&[l = lz_0 \to lez(lz_2, rc) , \\
&\ l = lz_1 \to \rho[s_1] \circ lez(lz_2, rc) , \\
&\ l = lz_2 \to \rho[b] \circ CF(\neg Q, \rho[s_2] \circ lez(lz_3, rc), lez(lz_1, rc)) , \\
&\ l = lz_3 \to I \\
&] \\
)&
\end{aligned}
$$

From these equations it follows that

- $\pi[\![x]\!]$

  $= lex(lx_0,F)$

  $= lex(lx_4,F)$

  $= lex(lx_1,I)$

  and

  $lex(lx_1,I)$

  $= \rho[\![b]\!] \circ CF(Q,\rho[\![s_1]\!] \circ lex(lx_1,I),\rho[\![s_2]\!])$.

- $\pi[\![y]\!]$

  $= ley(ly_0,F)$

  and

  $ley(ly_0,F)$

  $= \rho[\![b]\!] \circ CF(Q,\rho[\![s_1]\!] \circ ley(ly_0,F),\rho[\![s_2]\!])$.

- $\pi[\![z]\!]$

  $= lez(lz_0,F)$

  $= lez(lz_2,F)$

  and

  $lez(lz_2,F)$

  $= \rho[\![b]\!] \circ CF(\neg Q,\rho[\![s_2]\!],\rho[\![s_1]\!] \circ lez(lz_2,F))$

  $= \rho[\![b]\!] \circ CF(Q,\rho[\![s_1]\!] \circ lez(lz_2,F),\rho[\![s_2]\!])$.

We find that $\pi[\![x]\!]$, $\pi[\![y]\!]$ and $\pi[\![z]\!]$ all equal the least solution of the equation

$$W\colon\ W = \rho[\![b]\!] \circ CF(Q,\rho[\![s_1]\!] \circ W,\rho[\![s_2]\!]) ,$$

hence programs x, y and z are equivalent.
□

The following lemma states a property of the label environment le of program p which will be used in the derivation of version 2.

Lemma 5.14

Let le be as in definition 5.12.

$$le = (\lambda l \in \text{Label}, rc \in T \mid [l = l_0 \to \tau[s_0;s_1;\ldots;s_{n-1}](le,rc,I) ,$$
$$\ldots$$
$$l=l_{n-1} \to \tau[s_{n-1}](le,rc,I) ,$$
$$l = l_n \to I$$
$$])$$

□

Proof

Immediately from the fixed point property and definition 5.9.1.

□

5.3. Version 2: Introduction of program store

In this section we shall develop a semantics for program p based upon the representation of p in a program store, i.e. an array of instructions. This representation enables us to refer to arbitrary instructions by means of their index. As a consequence the label environment of version 1 can be eliminated. Version 2 makes use of a program store P, a label-to-address function la, and functions τ' and σ'. First we establish relations between these entities and those of version 1. Subsequently we derive from these relations and the definitions in version 1 an equation system in terms of P, la, τ', σ' and π alone.

Let us assume that the instruction sequences $s_0,\ldots,s_{n-1}$ of program p are stored consecutively in an array P(k: $0 \le k < N$) of instructions, where $N = (\underline{S}\ j: 0 \le j < n: \text{length}(s_j))$.

Let also be given a function la $\in$ Label $\to$ {0,...,N} such that:

- for all j: $0 \le j < n$: $la(l_j)$ is the index in P of the first instruction of s_j.
- $la(l_n) = N$.

It follows that

R1.1. for all j: $0 \le j < n$: $P(k: la(l_j) \le k < la(l_{j+1})) = s_j$.

R1.2. for all j: $0 \le j < n$: $P(k: la(l_j) \le k < N) = s_j;\ldots;s_{n-1}$.

Let le be as in definition 5.12. From R1.2 and lemma 5.14 we obtain

R2. $le = (\lambda l \in \text{Label}, rc \in T \mid$

$[l = l_0 \to \tau[P(k: la(l_0) \le k < N](le,rc,I)$,

...

$l = l_{n-1} \to \tau[P(k: la(l_{n-1}) \le k < N](le,rc,I)$,

$l = l_n \to I$

$]$

$)$

Definition 5.15 $\{\tau', \sigma'\}$

1. $\tau' \in \{0,\ldots,N\} \times T \to T$.

   for all j: $0 \leq j < N$, $rc \in T$:

   - $\tau'(j,rc) = \tau[\![P(k: j \leq k < N)]\!](le,rc,I)$.
   - $\tau'(N,rc) = I$.

2. $\sigma' \in$ Instruction $\to \{0,\ldots,N\} \times T \to T$.

   for all $i \in$ Instruction, j: $0 \leq j < N$, $rc \in T$:

   $\sigma'[\![i]\!](j,rc) = \sigma[\![i]\!](le,rc,\tau'(j+1,rc))$.

□

An immediate consequence of relation R2 and definition 5.15.1 is

R3. for all $l \in$ Label, $rc \in T$: $le(l,rc) = \tau'(la(l),rc)$.

Next, we derive a relation between τ' and σ'.
Let j: $0 \leq j < N$, $rc \in T$.

$\tau'(j,rc)$

= {definition 5.15.1}

$\tau[\![P(k: j \leq k < N)]\!](le,rc,I)$

= {instructions stored consecutively}

$\tau[\![P(j); P(k: j+1 \leq k < N)]\!](le,rc,I)$

= {definition 5.9}

$\rho[\![P(j)]\!] \circ \sigma[\![P(j)]\!](le,rc,\tau[\![P(k: j+1 \leq k < N)]\!](le,rc,I))$

= {definition 5.15.1}

$\rho[\![P(j)]\!] \circ \sigma[\![P(j)]\!](le,rc,\tau'(j+1,rc))$

= {definition 5.15.2}

$\rho[\![P(j)]\!] \circ \sigma'[\![P(j)]\!](j,rc)$.

The result of this derivation, together with the second part of definition 5.15.1, are summarized in:

R4. for all j: $0 \leq j < N$, $rc \in T$:

- $\tau'(j,rc) = \rho[\![P(j)]\!] \circ \sigma'[\![P(j)]\!](j,rc)$.
- $\tau'(N,rc) = I$.

For σ' we derive a relation R5, divided in cases. As the derivations of all cases have the same structure, we present the general pattern, followed by the results. The pattern is:

R5.m. $\sigma'[\![i_m]\!](j,rc)$

$= \{\text{definition 5.15.2}\}$

$E_1(\sigma,le,\tau',rc,j)$

$= \{\text{definition 5.8.m.2}\}$

$E_2(le,\tau',rc,j)$

$= \{R3\}$

$E_3(la,\tau',rc,j)$.

The results are:

R5.1. $\sigma'[\![LDA(a)]\!](j,rc) = \tau'(j+1,rc)$.

R5.2. $\sigma'[\![UJP(l)]\!](j,rc) = \tau'(la(l),rc)$.

R5.3. $\sigma'[\![FJP(l)]\!](j,rc) = CF(Q,\tau'(j+1,rc),\tau'(la(l),rc))$.

R5.4. $\sigma'[\![TJP(l)]\!](j,rc) = CF(\neg Q,\tau'(j+1,rc),\tau'(la(l),rc))$.

R5.5. $\sigma'[\![CSR(l)]\!](j,rc) = \tau'(la(l),\tau'(j+1,rc))$.

R5.6. $\sigma'[\![RET]\!](j,rc) = rc$.

Finally, from definition 5.12 and relation R3 we obtain

R6. $\pi[\![p]\!] = \tau'(la(l_0),F)$.

Taken together, relations R4, R5 and R6 are a semantics for p in terms of P, la, τ', σ' and π.

5.4. Version 3: Introduction of return stack

A short inspection of relations R4, R5 and R6 of version 2 reveals that each return continuation is either F or of the form $\tau'(k,rc)$, where $k \in \{0,\ldots,N\}$ and rc is an other return continuation. As a consequence we can characterize return continuations by means of a stack, represented here by a finite sequence over $\{0,\ldots,N\}$. We use a function f to define the return continuation represented by such a sequence.

Definition 5.16 {Stack}

$$\text{Stack} = \{0,\ldots,N\}^* .$$

□

Definition 5.17 {f}

$$f \in \text{Stack} \to T .$$

1. $f(< >) = F$.
2. $f(<j> \oplus s) = \tau'(j,f(s))$.

□

These definitions enable us to replace the equation system of version 2, in terms on P, 1a, τ', σ' and π, by an equation system in terms of P, 1a, τ'', σ'' and π, which is based on stacks instead of return continuations. We begin with definitions of τ'' and σ''.

Definition 5.18 {τ'', σ''}

1. $\tau'' \in \{0,\ldots,N\} \times \text{Stack} \to T$.

   for all j: $0 \le j \le N$, $s \in \text{Stack}$:

   $$\tau''(j,s) = \tau'(j,f(s)) .$$

2. $\sigma'' \in \text{Instruction} \to \{0,\ldots,N\} \times \text{Stack} \to T$.

   for all $i \in \text{Instruction}$, j: $0 \le j < N$, $rc \in T$:

   $$\sigma''[\![i]\!](j,s) = \sigma'[\![i]\!](j,f(s)) .$$

□

From definition 5.18 and relation R4 we obtain:

R7. for all j: $0 \leq j < N$, $s \in$ Stack:

- $\tau''(j,s) = \rho[\![P(j)]\!] \circ \sigma''[\![P(j)]\!](j,s)$.
- $\tau''(N,s) = I$.

For σ'' we derive relation R8 below, divided by cases. The derivations of the first four cases follow the pattern:

R8.m. $\sigma''[\![i_m]\!](j,s)$

= {definition 5.18.2}

$\sigma'[\![i_m]\!](j,f(s))$

= {R5.m}

$E_1(j,f(s),\tau',1a)$

= {definition 5.18.1}

$E_2(j,s,\tau'',1a)$.

For R8.5 the derivation is

$\sigma''[\![CSR(1)]\!](j,s)$

= {definition 5.18.2}

$\sigma'[\![CSR(1)]\!](j,f(s))$

= {R5.5}

$\tau'(1a(1),\tau'(j+1,f(s))$

= {definition 5.17.2}

$\tau'(1a(1),f(\langle j+1\rangle \oplus s))$

= {definition 5.18.1}

$\tau''(1a(1),\langle j+1\rangle \oplus s)$.

For R8.6 the derivation is

$\sigma''[\![RET]\!](j,s)$

= {definition 5.18.2}

$\sigma'[\![RET]\!](j,f(s))$

= {R5.6}

f(s)

= {definition 5.17}

if s = < > → F [] s = <j'> ⊕ s' → τ'(j',f(s')) fi

= {definition 5.18.1}

if s = < > → F [] s = <j'> ⊕ s' → τ"(j',s') fi .

Thus we obtain

R8.1. σ"⟦LDA(a)⟧(j,s) = τ"(j+1,s) .

R8.2. σ"⟦UJP(1)⟧(j,s) = τ"(la(1),s) .

R8.3. σ"⟦FJP(1)⟧(j,s) = CF(Q,τ"(j+1,s),τ"(la(1),s)) .

R8.4. σ"⟦TJP(1)⟧(j,s) = CF(¬Q,τ"(j+1,s),τ"(la(1),s)) .

R8.5. σ"⟦CSR(1)⟧(j,s) = τ"(la(1),<j+1> ⊕ s) .

R8.6. σ"⟦RET⟧(j,s) = if s = < > → F [] s = <j'> ⊕ s' → τ'(j',f(s')) fi .

Finally, for p we derive

π⟦p⟧

= {R6}

τ'(la(l_0), F)

= {definition 5.17.1}

τ'(la(l_0),f(< >))

= {definition 5.18.1}

τ"(la(l_0),< >) .

Hence

R9. π⟦p⟧ = τ"(la(l_0),< >) .

Relations R7, R8 and R9 characterize the semantics of p in terms of P, la, τ", σ" and π, using stacks instead of return continuations.

5.5. Version 4: Derivation of an interpreter

In this section we will derive an operational description of program p by means of an interpreter. Apart from the data variables M, A, B and Q, this interpreter will also use control variables, which are used for sequencing purposes. The control variables are:

P : array (k: $0 \leq k < N$) of Instruction ,

la: Label $\rightarrow$ {0,...,N} ,

ip: {0,...,N} ,

rs: Stack .

The variables P and la serve the same purpose as in version 2. The variable ip is the instruction pointer and indicates the location of an instruction to be interpreted. The variable rs is the return stack. For the interpreter P and la are to be considered as constants. All sequencing has to be performed by appropriate assignments to ip and rs.

We will code the interpreter in a slight variant of the source language, the semantics of which will be obvious. Our aim is to construct the interpreter in such a way that it has the same condition transformer as the program p to be interpreted. Relations R7, R8 and R9 will serve as guideline in the derivation.

To begin with, let us try to construct a repetition

DO = do B $\rightarrow$ S od

such that

R10. for all $s \in$ Stack, j: $0 \leq j < N$:

$$(ip,rs \leftarrow j,s) \circ wp(DO) = \tau''(j,s) .$$

If we succeed in doing so, it follows from relation R9 and the wp definition for assignment and sequential composition (definitions 3.37.3 and 3.37.4) that the program

ip,rs := $la(1_0)$, < > ; DO

has the same condition transformer as program p, so we may regard it as an operational description of p.

Applying the fixed point property for wp(DO) and some propositional calculus to R10 yields the following equivalent relation:

R11. for all $s \in Stack$, j: $0 \leq j \leq N$:

$$(ip,rs{\leftarrow}j,s) \circ (\lambda q \mid \neg B \wedge q \vee B \wedge wp(S)wp(DO)q) = \tau''(j,s) .$$

Separation of the case $j = N$ and application of R7 yields that R11 is equivalent to (R12.1 $\wedge$ R12.2), where

R12.1. for all $s \in Stack$:

$$(ip,rs{\leftarrow}N,s) \circ (\lambda q \mid \neg B \wedge q \vee B \wedge wp(S)wp(DO)q) = I .$$

R12.2. for all $s \in Stack$, j: $0 \leq j < N$:

$$(ip,rs{\leftarrow}j,s) \circ (\lambda q \mid \neg B \wedge q \vee B \wedge wp(S)wp(DO)q)$$
$$= \rho[\![P(j)]\!] \circ \sigma''[\![P(j)]\!](j,s) .$$

Relation R12.1 is satisfied by $B = (ip \neq N)$. Substitution in R12.2 yields

for all $s \in Stack$, j: $0 \leq j < N$:

$$(ip,rs{\leftarrow}j,s) \circ (\lambda q \mid ip = N \wedge q \vee ip \neq N \wedge wp(S)wp(DO)q)$$
$$= \rho[\![P(j)]\!] \circ \sigma''[\![P(j)]\!](j,s)$$

which can be simplified to

R13. for all $s \in Stack$, j: $0 \leq j < N$:

$$(ip,rs{\leftarrow}j,s) \circ wp(S) \circ wp(DO) = \rho[\![P(j)]\!] \circ \sigma''[\![P(j)]\!](j,s) .$$

It follows that we should look for a program

$$DO = \underline{do}\ ip \neq N \rightarrow S\ \underline{od}$$

where, under the assumption that R10 holds - this is in fact the induction hypothesis for fixed point induction - the statement S should satisfy relation R13.

Obviously, relation R13 depends on the value of P(j) for various j. Let us therefore rewrite R13 as:

R14. for all $s \in Stack$, $j: 0 \leq j < N$:

$$(ip,rs{\leftarrow}j,s) \circ wp(S) \circ wp(DO)$$
$$= (\lambda q \mid [\bigvee i \mid i \in Instruction \mid P(j) = i \wedge \rho[\![i]\!] \sigma''[\![i]\!](j,s)q]) .$$

This relation is satisfied by

$$wp(S) = (\lambda q \mid [\bigvee i \mid i \in Instruction \mid P(ip) = i \wedge wp(T_i)q])$$

provided we can find statements T_i such that

R15. for all $i \in Instruction$, $s \in Stack$, $j: 0 \leq j < N$:

$$(ip,rs{\leftarrow}j,s) \circ wp(T_i) \circ wp(DO) = \rho[\![i]\!] \circ \sigma''[\![i]\!](j,s) .$$

Let us denote the set Instruction by the k element set $\{i_1,\ldots,i_k\}$ for a while. As wp(S) is the condition transformer of an alternative statement with mutually exclusive guards $P(ip) = i_k$, it follows that we should look for a program

$$DO = \underline{do}\ ip \neq N \rightarrow \underline{if}\ P(ip) = i_1 \rightarrow T_{i_1}$$
$$[]\ \ldots$$
$$[]\ P(ip) = i_k \rightarrow T_{i_k}$$
$$\underline{fi}$$
$$\underline{od}$$

where, under the assumption that R10 holds, the statements T_i should satisfy relation R15.

Let us restrict ourselves to statements T_i of the form

$$X_i;\ Y_i$$

where the statements X_i do not contain assignments to the variables ip and rs. In that case relation R15 may be rewritten as:

R16. for all $i \in Instruction$, $s \in Stack$, $j: 0 \leq j < N$:

$$wp(X_i) \circ (ip,rs{\leftarrow}j,s) \circ wp(Y_i) \circ wp(DO) = \rho[\![i]\!] \circ \sigma''[\![i]\!](j,s)$$

which is implied by (R17 $\wedge$ R18), where

R17. for all $i \in Instruction$: $wp(X_i) = \rho[\![i]\!]$.

R18. For all $i \in$ Instruction, $s \in$ Stack, j: $0 \leq j < N$:

$$(ip,rs\leftarrow j,s) \circ wp(Y_i) \circ wp(DO) = \sigma''[\![i]\!](j,s) .$$

Now it is time to consider the individual instructions. Let us first deal with the solutions of R17.

- If i = LDA(a), then by definition 5.8.1.1: $\rho[\![i]\!] = (A \leftarrow M(a))$, hence $X_i = A := M(a)$.
- If i is a singular instruction, then by definitions 5.8.2.1-5.8.6.1: $\rho[\![i]\!] = I$, hence X_i = skip.

Next we derive, under the assumption that R10 holds, statements Y_i satisfying R18. For two representative cases we give full derivations. The other cases are similar.

case i = LDA(a)

$\sigma''[\![i]\!](j,s)$

= {R8.1}

$\tau''(j+1,s)$

= {R10}

$(ip,rs\leftarrow j+1,s) \circ wp(DO)$

= {property of substitution}

$(ip,rs\leftarrow j,s) \circ (ip\leftarrow ip+1) \circ wp(DO)$

= {definition wp}

$(ip,rs\leftarrow j,s) \circ wp(ip := ip+1) \circ wp(DO)$

hence Y_i = ip := ip+1.

case i = CSR(1)

$\sigma''[\![i]\!](j,s)$

= {R8.5}

$\tau''(1a(1),\langle j+1\rangle \oplus s)$

= {R10}

$$(ip,rs \leftarrow la(l), \langle j+1 \rangle \oplus s) \circ wp(DO)$$

$$= \{\text{property of substitution}\}$$

$$(ip,rs \leftarrow j,s) \circ (ip,rs \leftarrow la(l), \langle j+1 \rangle \oplus s) \circ wp(DO)$$

$$= \{\text{definition wp}\}$$

$$(ip,rs \leftarrow j,s) \circ wp(ip,rs := la(l), \langle j+1 \rangle \oplus s) \circ wp(DO)$$

hence $Y_i = ip,rs := la(l), \langle j+1 \rangle \oplus s$.

By similar derivations we find

case i = UJP(1): $Y_i = ip := la(l)$.

case i = FJP(1): $Y_i = \underline{if}\ Q \rightarrow ip := ip+1 \ [] \ \neg Q \rightarrow ip := la(l)\ \underline{fi}$

case i = TJP(1): $Y_i = \underline{if}\ \neg Q \rightarrow ip := ip+1 \ [] \ Q \rightarrow ip := la(l)\ \underline{fi}$

case i = RET : $Y_i = \underline{if}\ s = \langle j' \rangle \oplus s' \rightarrow ip,rs := j',s'\ \underline{fi}$

With the derivation of the statements X_i and Y_i we have completed the derivation of the interpreter. Combination of the code fragments yields:

```
ip,rs := la(l0), < >;
do ip ≠ N →
   if P(ip) = LDA(a) → A := M(a); ip := ip+1
    [] P(ip) = UJP(1) → skip; ip := la(l)
    [] P(ip) = FJP(1) → skip;
                        if Q → ip := ip+1 [] ¬ Q → ip := la(l) fi
    [] P(ip) = TJP(1) → skip;
                        if ¬ Q → ip := ip+1 [] Q → ip := la(l) fi
    [] P(ip) = CSR(1) → skip; ip,rs := la(1),<j+1> ⊕ rs
    [] P(ip) = RET    → skip;
                        if s = <j'> ⊕ s' → ip,rs := j',s' fi
   fi
od
```

CHAPTER 6
EPILOGUE

We conclude this thesis with a short summary and evaluation of the results obtained in preceding chapters.

In chapter 2 we have developed a variant of attribute grammars which is a self-contained formal system free of implementation bias. Nevertheless, the various components of this system can be used directly as compiler specifications, from which compilers can be derived by means of existing techniques for attribute evaluation and data structure implementation. The grammar for the source language is rather compact: it has only 33 grammar rules.

We have extended the predicate transformer method to a genuine definition method by providing it with a firm foundation and extending it to language constructs other than statements. In this respect the lattice theory of section 3.1 has been of great value. The general framework provided by this theory has helped in structuring definitions and in separating general lattice-theoretical properties from properties particular to certain language constructs.

We have developed semantics and proof rules for recursive procedures, both with and without parameters, constructs the formal treatment of which in the literature has often been problematic. We have succeeded in doing so by separation of the various aspects of procedures and by making design choices leading to simple semantics, e.g. with respect to parameter mechanisms and initialization requirements.

The background of our work has made it necessary to consider both syntax and semantics. On the one hand, taking into account context-dependent properties has complicated various derivations, and we feel that there is room for improvement in this respect. On the other hand, it has enabled us to derive rather simple conditions for the applicability of certain proof rules, e.g. with respect to scope of variables and disjointness of parameters, a notorious problem area.

We have shown that it is possible to give a manageable non-operational definition of machine instructions, and that an implementation of these instructions can be derived systematically. The latter derivation also gives an impression of the way the formalisms developed in this thesis will be put to work in the derivation of a translation from SL to TL.

APPENDIX A
PROOF OF SOME LEMMAS

Proof of lemma 3.20.2

$(C,\sqsubseteq)$ is a uccl

$\Rightarrow$ {definition 3.5, R, S countable}

$(\underline{A}\ i \in R,\ j \in S \mid x_{ij} \sqsubseteq \bigsqcup_{j\in S} x_{ij} \wedge \bigsqcup_{j\in S} x_{ij} \sqsubseteq \bigsqcup_{i\in R} \bigsqcup_{j\in S} x_{ij})$

$\Rightarrow$ {transitivity $\sqsubseteq$, definition 3.2}

$(\underline{A}\ j \in S \mid \bigsqcup_{i\in R} x_{ij} \sqsubseteq \bigsqcup_{i\in R} \bigsqcup_{j\in S} x_{ij})$

$\Rightarrow$ {definition 3.2}

$$\bigsqcup_{j\in S} \bigsqcup_{i\in R} x_{ij} \sqsubseteq \bigsqcup_{i\in R} \bigsqcup_{j\in S} x_{ij} .$$

Also, by symmetry:

$$\bigsqcup_{j\in S} \bigsqcup_{i\in R} x_{ij} \sqsupseteq \bigsqcup_{i\in R} \bigsqcup_{j\in S} x_{ij} .$$

Hence, by antisymmetry of $\sqsubseteq$:

$$\bigsqcup_{i\in R} \bigsqcup_{j\in S} x_{ij} = \bigsqcup_{j\in S} \bigsqcup_{i\in R} x_{ij} .$$

□

Proof of lemma 3.21

1. $\langle x_i \rangle_{i=0}^{\infty}$ and $\langle y_i \rangle_{i=0}^{\infty}$ ascending chains

$= (\underline{A}\ i \mid i \geq 0 \mid x_i \sqsubseteq x_{i+1} \wedge y_i \sqsubseteq y_{i+1})$

$\Rightarrow$ {definition $\sqcap$}

$(\underline{A}\ i \mid i \geq 0 \mid x_i \sqcap y_i \sqsubseteq x_{i+1} \wedge x_i \sqcap y_i \sqsubseteq y_{i+1})$

$=$ {definition $\sqcap$}

$(\underline{A}\ i \mid i \geq 0 \mid x_i \sqcap y_i \sqsubseteq x_{i+1} \sqcap y_{i+1})$

$= \langle x_i \sqcap y_i \rangle_{i=0}^{\infty}$ is an ascending chain.

2. Let $z = \bigsqcup_{i=0}^{\infty} (x_i \sqcap y_i)$, $x = \bigsqcup_{i=0}^{\infty} x_i$, $y = \bigsqcup_{i=0}^{\infty} y_i$.

We first prove

1. $z \sqsubseteq x \sqcap y$, and subsequently the stronger assertion
2. $z = x \sqcap y$.

1. true

= {definition $\sqcap$}

$(\underline{A}\ i \mid i \geq 0 \mid x_i \sqcap y_i \sqsubseteq x_i \wedge x_i \sqcap y_i \sqsubseteq y_i)$

$\Rightarrow$ {definition $\sqcup$}

$(\underline{A}\ i \mid i \geq 0 \mid x_i \sqcap y_i \sqsubseteq \bigsqcup_{i=0}^{\infty} x_i \wedge x_i \sqcap y_i \sqsubseteq \bigsqcup_{i=0}^{\infty} y_i)$

= {definition $\sqcap$}

$(\underline{A}\ i \mid i \geq 0 \mid x_i \sqcap y_i \sqsubseteq (\bigsqcup_{i=0}^{\infty} x_i) \sqcap (\bigsqcup_{i=0}^{\infty} y_i))$

= {definition $\sqcup$}

$\bigsqcup_{i=0}^{\infty} (x_i \sqcap y_i) \sqsubseteq (\bigsqcup_{i=0}^{\infty} x_i) \sqcap (\bigsqcup_{i=0}^{\infty} y_i)$

= {definition x, y, z}

$z \sqsubseteq x \sqcap y$.

2. $z \sqsubseteq x \sqcap y$

= {definition 3.1}

$(z = x \sqcap y) \vee (z \sqsubset x \sqcap y)$

= {definition x, y, $\sqcap$}

$(z = x \sqcap y) \vee (z \sqsubset \bigsqcup_{i=0}^{\infty} x_i \wedge z \sqsubset \bigsqcup_{i=0}^{\infty} y_i)$

$\Rightarrow$ {definition $\sqcup$}

$(z = x \sqcap y) \vee ((\underline{E}\ i \mid i \geq 0 \mid z \sqsubset x_i) \wedge (\underline{E}\ i \mid i \geq 0 \mid z \sqsubset y_i))$

$\Rightarrow$ {$\langle x_i \rangle_{i=0}^{\infty}$ and $\langle y_i \rangle_{i=0}^{\infty}$ ascending}

$(z = x \sqcap y) \vee (\underline{E}\ i \mid i \geq 0 \mid z \sqsubset x_i \wedge z \sqsubset y_i)$

= {definition $\sqcap$}

$(z = x \sqcap y) \vee (\underline{E}\ i \mid i \geq 0 \mid z \sqsubset x_i \sqcap y_i)$

= {definition $\sqcup$}

$$(z = x \sqcap y) \vee (z \sqsubset \bigsqcup_{i=0}^{\infty} (x_i \sqcap y_i))$$

$$= \{\text{definition } z\}$$

$$(z = x \sqcap y) \vee (z \sqsubset z)$$

$$= (z = x \sqcap y) .$$

□

Proof of lemma 3.22

By induction on the size of S.

1. For $S = \emptyset$ the equality holds, because $\sqcap \emptyset = \tau_c$.

2. Ind. hyp.: let for some finite set S'

$$\prod_{i \in S'} \bigsqcup_{j=0}^{\infty} x_{ij} = \bigsqcup_{j=0}^{\infty} \prod_{i \in S'} x_{ij} .$$

Let $1 \notin S'$, and $S = \{1\} \cup S'$.

$$\prod_{i \in S} \bigsqcup_{j=0}^{\infty} x_{ij}$$

$$= \{S = \{1\} \cup S'\} \ (\bigsqcup_{j=0}^{\infty} x_{ij}) \sqcap (\prod_{i \in S'} \bigsqcup_{j=0}^{\infty} x_{ij})$$

$$= \{\text{ind. hyp.}\} \quad (\bigsqcup_{j=0}^{\infty} x_{ij}) \sqcap (\bigsqcup_{j=0}^{\infty} \prod_{i \in S'} x_{ij})$$

$$= \{\text{lemma 3.21.2}\} \ \bigsqcup_{j=0}^{\infty} (x_{ij} \sqcap \prod_{i \in S'} x_{ij})$$

$$= \{S = \{1\} \cup S'\} \ \bigsqcup_{j=0}^{\infty} \prod_{i \in S} x_{ij} .$$

□

Proof of lemma 3.25

1. $$g \circ \left(\bigsqcup_{i=0}^{\infty} h_i \right)$$

$$= \{\text{definition } \lfloor D \rfloor\} \ g \circ \left(\lambda x \mid \bigsqcup_{i=0}^{\infty} h_i(x) \right)$$

$$= \{\text{functional composition}\} \ \left(\lambda x \mid g\left(\bigsqcup_{i=0}^{\infty} h_i(x) \right) \right)$$

$$= \{g \in D\} \ \left(\lambda x \mid \bigsqcup_{i=0}^{\infty} g(h_i(x)) \right)$$

$$= \{\text{definition } \lfloor D \rfloor\} \ \bigsqcup_{i=0}^{\infty} (\lambda x \mid g(h_i(x)))$$

$$= \{\text{functional composition}\} \quad \lfloor\!\underline{D}\!\rfloor_{i=0}^{\infty} (g \circ h_i) \ .$$

2. $\left(\lfloor\!\underline{D}\!\rfloor_{i=0}^{\infty} g_i\right) \circ \left(\lfloor\!\underline{D}\!\rfloor_{i=0}^{\infty} h_j\right)$

$$= \{\text{functional composition}\} \quad \left(\lambda x \mid \left(\lfloor\!\underline{D}\!\rfloor_{i=0}^{\infty} g_i\right)\left(\left(\lfloor\!\underline{D}\!\rfloor_{j=0}^{\infty} h_j\right)(x)\right)\right)$$

$$= \{\text{definition } \lfloor\!\underline{D}\!\rfloor\} \quad \left(\lambda x \mid \lfloor\!\underline{C}\!\rfloor_{i=0}^{\infty} g_i \left(\lfloor\!\underline{C}\!\rfloor_{j=0}^{\infty} h_j(x)\right)\right)$$

$$= \{g_i \in D\} \quad \left(\lambda x \mid \lfloor\!\underline{C}\!\rfloor_{i=0}^{\infty} \lfloor\!\underline{C}\!\rfloor_{j=0}^{\infty} g_i(h_j(x))\right)$$

$$= \{\text{lemma 3.20.1}\} \quad \left(\lambda x \mid \lfloor\!\underline{C}\!\rfloor_{k=0}^{\infty} g_k(h_k(x))\right)$$

$$= \{\text{definition } \lfloor\!\underline{D}\!\rfloor\} \quad \lfloor\!\underline{D}\!\rfloor_{k=0}^{\infty} (\lambda x \mid g_k(h_k(x)))$$

$$= \{\text{functional composition}\} \quad \lfloor\!\underline{D}\!\rfloor_{k=0}^{\infty} (g_k \circ h_k) \ .$$

□

APPENDIX B
COLLECTED DEFINITION OF THE SOURCE LANGUAGE

Syntax

{Collected from sections 2.3.2, 4.1.1, 4.2.1, 4.4.1. The functions USE, ASSN, and INIT have been incorporated into the attribute grammar. The corresponding attribute variables have names beginning with the letters u, a, and i, respectively. Their domain is *Nst*, which corresponds to sets of names.}

Domains

{*Bool*, *Int*, *Name*, *Names*, *Nst*, *Prio*, *Type*, *Types*, *Decs*, *Env*, *Penv*}

Attribute variables

n, n_1, n_2: *Name*;
ns, ns_0, ns_1, ns_2: *Names*;
u, u_0, u_1, u_2, ug, ug_0, ug_1, ug_2, us, us_0, us_1, us_2, a, a_0, a_1, a_2,
i, i_0, i_1, i_2: *Nst*;
p, p_0, p_1, p_2: *Prio*;
t, t_0, t_1, t_2: *Type*;
ts, ts_0, ts_1, ts_2, ts_3, ts_4: *Types*;
d, d_0, d_1, d_2: *Decs*;
e, e_0, e_1: *Env*;
pe, pe_0, pe_1, pe_2, pe_3: *Penv*.

Operations on *Name*

Name = Letter (Letter ∪ Digit)*.

Operations on *Names*

$[\cdot]_N$: *Name* → *Names*
$\cdot \cup_N \cdot$: *Names* × *Names* → *Names*
$\cdot \in_N \cdot$: *Name* × *Names* → *Bool*
$\#_N(\cdot,\cdot)$: *Name* × *Names* → *Int*

$n_1 \underline{in}_N [n_2]_N = (n_1 = n_2)$

$n_1 \underline{in}_N (ns_1 \textcircled{N} ns_2) = (n_1 \underline{in}_N ns_1) \vee (n_1 \underline{in}_N ns_2)$

$\#_N (n_1,[n_2]_N) = \underline{if}\ n_1 = n_2 \rightarrow 1 \ [] \ n_1 \neq n_2 \rightarrow 0\ \underline{fi}$

$\#_N (n_1,ns_1 \textcircled{N} ns_2) = \#_N (n,ns_1) + \#_N (n,ns_2)$

Operations on *Nst*

Nst = set of *Name*. Details of set axioms omitted.

Operations on *Prio*

Prio = {e $\in$ *Int* | the integer value corresponding to e is an element of {1,...,7}} .

Operations on *Type*

Type = Typesym.

Operations on *Types*

$[\cdot]_T$: *Type* $\rightarrow$ *Types*

$\cdot \oplus_T \cdot$: *Types* $\times$ *Types* $\rightarrow$ *Types*

Mts : *Names* $\times$ *Type* $\rightarrow$ *Types*

$(ts_1 \oplus_T ts_2) \oplus_T ts_3 = ts_1 \oplus_T (ts_2 \oplus_T ts_3)$

$Mts([n]_N,t) = [t]_T$

$Mts(ns_1 \textcircled{N} ns_2,t) = Mts(ns_1,t) \oplus Mts(ns_2,t)$

Operations on *Decs*

$[\cdot,\cdot]_D$: *Names* $\times$ *Type* $\rightarrow$ *Decs*

$\cdot \textcircled{D} \cdot$: *Decs* $\times$ *Decs* $\rightarrow$ *Decs*

$(\cdot,\cdot) \underline{in}_D \cdot$: *Name* $\times$ *Type* $\times$ *Decs* $\rightarrow$ *Bool*

$\#_D (\cdot,\cdot)$: *Name* $\times$ *Decs* $\rightarrow$ *Int*

$(n,t_1) \underline{in}_D [ns,t_2]_D = n \underline{in}_N ns \wedge (t_1 = t_2)$

$(n,t) \underline{in}_D (d_1 \textcircled{D} d_2) = ((n,t) \underline{in}_D d_1) \vee ((n,t) \underline{in}_D d_2)$

$\#_D\ (n,[ns,t]_D) = \#_N\ (n,ns)$

$\#_D\ (n,d_1 \textcircled{D} d_2) = \#_D\ (n,d_1) + \#_D\ (n,d_2)$

Operations on *Env*

Empty : *Env*

Ext(•,•) : *Env* * *Decs* → *Env*

(•,•) $\underline{in}_E$ • : *Name* * *Type* * *Env* → *Bool*

(n,t) $\underline{in}_E$ Empty = false

(n,t) $\underline{in}_E$ Ext(e,d) = (n,t) $\underline{in}_D$ d ∨ ($\#_D$ (n,d) = 0 ∧ (n,t) $\underline{in}_E$ e)

Operations on *Penv*

Pempty : *Penv*

$[\bullet,\bullet,\bullet]_P$: *Name* * *Types* * *Types* → *Penv*

• $\textcircled{P}$ • : *Penv* * *Penv* → *Penv*

$\#_P(\bullet,\bullet)$: *Name* * *Penv* → *Int*

(•,•,•) $\underline{in}_P$ • : *Name* * *Types* * *Types* * *Penv* → *Bool*

$\#_P$ (n,Pempty) = 0

$\#_P\ (n_1,[n_2,ts_1,ts_2]_P) = \underline{if}\ n_1 = n_2 \rightarrow 1\ []\ n_1 \neq n_2 \rightarrow 0\ \underline{fi}$

$\#_P\ (n,pe_1 \textcircled{P} pe_2) = \#_P\ (n,pe_1) + \#_P\ (n,pe_2)$

$(n,ts_1,ts_2)\ \underline{in}_P$ Pempty = false

$(n_1,ts_1,ts_2)\ \underline{in}_P\ [n_2,ts_3,ts_4]_P = (n_1 = n_2 \wedge ts_1 = ts_3 \wedge ts_2 = ts_4)$

$(n,ts_1,ts_2)\ \underline{in}_P\ pe_1 \textcircled{P} pe_2 = ((n,ts_1,ts_2)\ \underline{in}_P\ pe_1) \vee ((n,ts_1,ts_2)\ \underline{in}_P\ pe_2)$

Nonterminals

V_N = {Abstr <*Penv*,*Types*,*Types*> ,
Block <*Penv*,*Env*,*Nst*,*Nst*,*Nst*> ,
Con <*Type*> ,
Decs <*Decs*> ,
Dop <*Prio*,*Type*,*Type*,*Type*> ,
Expr <*Env*,*Prio*,*Type*,*Nst*> ,
Exprs <*Env*,*Types*,*Nst*> ,

Gcs <*Penv*,*Env*,*Nst*,*Nst*,*Nst*,*Nst*> ,
Id <*Name*> ,
Ids <*Names*> ,
Mop <*Type*,*Type*> ,
Pdecs <*Decs*,*Types*> ,
Procdecs <*Penv*,*Penv*> ,
Prog ,
Stat <*Penv*,*Env*,*Nst*,*Nst*,*Nst*> ,
Type <*Type*> ,
Var <*Env*,*Name*,*Type*> ,
Vars <*Env*,*Names*,*Types*,*Nst*>
}

Terminals

Letter = {"a",...,"z"}
Digit = {"0",...,"9"}
Op1 = {"+","-","¬"}
Op2 = {"*","+","-","=","≠","<","≤",">","≥","∧","∨","⇒","⇔"}

Typesym = {"int","bool"}
Consym = {"true","false"}
Statsym = {"skip","abort"}
Sym = {"|[","]|","|",",",":",";", "[]","→",":=","(",")",
"if","fi","do","od","var","con","res"}

V_T = Letter ∪ Digit ∪ Op1 ∪ Op2 ∪ Typesym ∪ Consym ∪ Statsym ∪ Sym .

Start symbol

Prog.

Pseudo terminals

{Id <*Name*>, Dop <*Prio*,*Type*,*Type*,*Type*>, Mop <*Type*,*Type*>, Con <*Type*>,
Type <*Type*>}

For all n ∈ *Name*:

L(Id <n>) = {n} \ (Typesym ∪ Consym ∪ Statsym) .

L(Dop <1,bool,bool,bool>) = {"⇒","⇔"}
L(Dop <2,bool,bool,bool>) = {"∨"}
L(Dop <3,bool,bool,bool>) = {"∧"}
L(Dop <4,bool,int,int>) = {"=","≠","<","≤",">","≥"}
L(Dop <5,int,int,int>) = {"+","−"}
L(Dop <6,int,int,int>) = {"*"}

L(Mop <int,int>) = {"+","−"}
L(Mop <bool,bool>) = {"¬"}

L(Con <int>) = Digit$^+$
L(Con <bool>) = Consym

L(Type <int>) = {"int"}
L(Type <bool>) = {"bool"}

Grammar rules

1. Prog ::=
   Block <pe,e,u,a,i> ■
   pe = Pempty
   e = Empty

2. Prog ::=
   Procdecs <pe,pe> | Block <pe,e,u,a,i> ■
   e = Empty
   (A n: *Name* | $\#_P$ (n,pe) ≤ 1)

3. Procdecs <pe_0,pe_1> ::=
   Procdecs <pe_0,pe_2> , Procdecs <pe_0,pe_3> ■
   pe_1 = pe_2 ⓟ pe_3

4. Procdecs <pe_0,pe_1> ::=
   Id <n> = Abstr <pe_0,ts_1,ts_2> ■
   pe_1 = $[n,ts_1,ts_2]_P$

5. Abstr <pe,ts_1,ts_2> ::=

(con Pdecs <d_1,ts_1> ; res Pdecs <d_2,ts_2>

| Stat <pe,e,u,a,i>) ■

(A n: *Name* | $\#_D$ (n,d_1 ⩒ d_2) + $\#_P$ (n,pe) ≤ 1)

e = Ext(Empty,d_1 ⩒ d_2)

¬(E n: *Name* | $\#_D$ (n,d_1) ≠ 0 ∧ n ∈ a)

(A n: *Name* | $\#_D$ (n,d_2) ≠ 0 ⇔ n ∈ i)

6. Pdecs <d_0,ts_0> ::=

Pdecs <d_1,ts_1> , Pdecs <d_2,ts_2> ■

d_0 = d_1 ⩒ d_2

ts_0 = ts_1 ⊕ ts_2

7. Pdecs <d,ts> ::=

Ids <ns> : Type <t> ■

d = $[ns,t]_D$

ts = Mts(ns,t)

8. Block <pe,e_0,u_0,a_0,i_0> ::=

|[var Decs <d> | Stat <pe,e,e_1,u_1,a_1,i_1>

]| ■

(A n: *Name* | $\#_D$ (n,d) + $\#_P$ (n,pe) ≤ 1)

e_1 = Ext(e_0,d)

(A n: *Name* | $\#_D$ (n,d) ≠ 0 ⇒ (n ∈ i_1 ∨ n ∉ u_1))

u_0 = u_1 \ {n: *Name* | $\#_D$ (n,d) ≠ 0}

a_0 = a_1 \ {n: *Name* | $\#_D$ (n,d) ≠ 0}

i_0 = i_1 \ {n: *Name* | $\#_D$ (n,d) ≠ 0}

9. Decs <d_0> ::=

Decs <d_1> , Decs <d_2> ■

d_0 = d_1 ⩒ d_2

10. Decs <d> ::=

Ids <ns> : Type <t> ■

d = $[ns,t]_D$

11. Ids <ns_0> ::=

Ids <ns_1> , Ids <ns_2> ■

ns_0 = ns_1 ⩒ ns_2

12. Ids <ns> ::=

    Id <n> ■

    ns = $[n]_N$

13. Stat <pe,e,u,a,i> ::=

    abort ■

    u = ∅

    a = ∅

    i = ∅

14. Stat <pe,e,u,a,i> ::=

    skip ■

    u = ∅

    a = ∅

    i = ∅

15. Stat <pe,e,u_0,a_0,i_0> ::=

    Vars <e,ns,ts,a_0> := Exprs <e,ts,u_0> ■

    (A n: *Name* | $\#_N$ (n,ns) ≤ 1)

    $i_0 = a_0 \setminus u_0$

16. Stat <pe,e,u_0,a_0,i_0> ::=

    Stat <pe,e,u_1,a_1,i_1> ; Stat <pe,e,u_2,a_2,i_2> ■

    $u_0 = u_1 \cup u_2$

    $a_0 = a_1 \cup a_2$

    $i_0 = i_1 \cup (i_2 \setminus u_1)$

17. Stat <pe,e,u,a,i_0> ::=

    if Gcs <pe,e,ug,us,a,i_1> fi ■

    u = ug ∪ us

    $i_0 = i_1 \setminus ug$

18. Stat <pe,e,u,a,i_0> ::=

    do Gcs <pe,e,ug,us,a,i_1> od ■

    u = ug ∪ us

    i_0 = ∅

19. Stat <pe,e,u,a,i> ::=

    Block <pe,e,u,a,i> ■

20. Stat $\langle pe,e,u_0,a_0,i_0 \rangle$ ::=
    Abstr $\langle pe,ts_1,ts_2 \rangle$ (Exprs $\langle e,ts_1,u_0 \rangle$;
    Vars $\langle e,ns,ts_2,a_0 \rangle$) ■
    ($\underline{A}$ n: *Name* | $\#_N$ (n,ns) ≤ 1)
    $i_0 = a_0 \setminus u_0$

21. Stat $\langle pe,e,u_0,a_0,i_0 \rangle$ ::=
    Id $\langle n \rangle$ (Exprs $\langle e,ts_1,u_0 \rangle$;
    Vars $\langle e,ns,ts_2,a_0 \rangle$) ■
    ($\underline{A}$ n: *Name* | $\#_N$ (n,ns) ≤ 1)
    (n,ts_1,ts_2) $\underline{in}_P$ pe
    $i_0 = a_0 \setminus u_0$

22. Vars $\langle e,ns_0,ts_0,a_0 \rangle$ ::=
    Vars $\langle e,ns,ts_1,a_1 \rangle$, Vars $\langle e,ns_2,ts_2,a_2 \rangle$ ■
    $ns_0 = ns_1 \textcircled{N} ns_2$
    $ts_0 = ts_1 \oplus_T ts_2$
    $a_0 = a_1 \cup a_2$

23. Vars $\langle e,ns,ts,a \rangle$ ::=
    Var $\langle e,n,t \rangle$ ■
    $ns = [n]_N$
    $ts = [t]_T$
    $a = \{n\}$

24. Exprs $\langle e,ts_0,u_0 \rangle$ ::=
    Exprs $\langle e,ts_1,u_1 \rangle$, Exprs $\langle e,ts_2,u_2 \rangle$ ■
    $ts_0 = ts_1 \oplus_T ts_2$
    $u_0 = u_1 \cup u_2$

25. Exprs $\langle e,ts,u \rangle$ ::=
    Expr $\langle e,p,t,u \rangle$ ■
    $ts = [t]_T$

26. Expr $\langle e,p_0,t_0,u_0 \rangle$ ::=
    Expr $\langle e,p_1,t_1,u_1 \rangle$ Dop $\langle p_0,t_0,t_1,t_2 \rangle$ Expr $\langle e,p_2,t_2,u_2 \rangle$ ■
    $p_0 \leq p_1$
    $p_0 < p_2$
    $u_0 = u_1 \cup u_2$

27. Expr <e,p_0,t_0,u> ::=
    Mop <t_0,t_1> Expr <e,p_1,t_1,u> ■
    p_0 = 7
    p_1 = 7

28. Expr <e,p_0,t,u> ::=
    (Expr <e,p_1,t,u>) ■
    p_0 = 7

29. Expr <e,p,t,u> ::=
    Var <e,n,t> ■
    p = 7
    u = {n}

30. Expr <e,p,t,u> ::=
    Con <t> ■
    p = 7
    u = ∅

31. Var <e,n,t> ::=
    Id <n> ■
    (n,t) $\underline{\text{in}}_E$ e

32. Gcs <pe,e,ug_0,us_0,a_0,i_0> ::=
    Gcs <pe,e,ug_1,us_1,a_1,s_1> ▯ Gcs <pe,e,ug_2,us_2,a_2,i_2> ■
    ug_0 = ug_1 ∪ ug_2
    us_0 = us_1 ∪ us_2
    a_0 = a_1 ∪ a_2
    i_0 = i_1 ∩ i_2

33. Gcs <pe,e,ug,us,a,i> ::=
    Expr <e,p,t,ug> → Stat <pe,e,us,a,i> ■
    t = bool

Semantics

{Collected from definitions 4.46, 4.47, 4.48, 4.49.}

Definition {wp'}
For all e ∈ *Env* the function

$$wp'_e \in L(\text{Stat } \langle Penv, e, Nst, Nst, Nst\rangle) \to \Delta \to T_e$$

is defined by

1. $wp'_e(\text{abort})\delta = (\lambda q \in C_e \mid \text{false})$
2. $wp'_e(\text{skip})\delta = (\lambda q \in C_e \mid q)$
3. $wp'_e(v := E)\delta = (v \leftarrow E)$
4. $wp'_e(S_1;S_2)\delta = (wp'_e(S_1)\delta) \circ (wp'_e(S_2)\delta)$
5. $wp'_e(\underline{\text{if}}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{\text{fi}})\delta =$
   $= (\lambda q \in C_e \mid [\bigvee i \mid 1{\le}i{\le}n \mid B_i] \wedge [\bigwedge i \mid 1{\le}i{\le}n \mid B_i \Rightarrow (wp'_e(S_i)\delta)q])$
6. $wp'_e(\underline{\text{do}}\ B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n\ \underline{\text{od}})\delta = \mu F$,
   where $F = (\lambda f \in C_e \to_{uc} C_e \mid$
   $(\lambda q \in C_e \mid ([\bigvee i \mid 1{\le}i{\le}n \mid B_i] \vee q)$
   $\wedge [\bigwedge i \mid 1{\le}i{\le}n \mid B_i \Rightarrow (wp'_e(S_i)\delta)\ f\ q]))$
7. $wp'_e(A(E;v))\delta = pwp'_e(A)(\delta)(E,v)$
8. $wp'_e(p(E;v))\delta = pwp'_e(p)(\delta)(E,v)$
9. $wp'_e(\ |[\ \underline{\text{var}}\ x{:}\ t \mid S\ ]|\ ) = (\lambda q \in C_e \mid (wp'_{e_1}((x \leftarrow y)S)\delta)q)$,
   where $y \in \underline{Name}$ such that $\vdash$ new (y,e) ,
   $e_1 = \text{Ext}(e,[y,t]_D)$.

□

Definition {wlp'}

For all e ∈ *Env* the function

$$wlp'_e \in L(\text{Stat } \langle Penv, e, Nst, Nst, Nst\rangle) \to \Delta \to T_e$$

is defined by

1. $wlp'_e(\text{abort})\delta = (\lambda q \in C_e \mid \text{true})$
2. $wlp'_e(\text{skip})\delta = (\lambda q \in C_e \mid q)$
3. $wlp'_e(v := E)\delta = (v \leftarrow E)$
4. $wlp'_e(S_1;S_2)\delta = (wlp'_e(S_1)\delta)\circ(wlp'_e(S_2)\delta)$
5. $wlp'_e(\underline{\text{if}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{fi}}) =$
   $= (\lambda q \in C_e \mid [\bigwedge i \mid 1\le i\le n \mid B_i \Rightarrow (wlp'_e(S_i)\delta)q])$
6. $wlp'_e(\underline{\text{do}}\ B_1 \to S_1 \ [] \ \ldots \ [] \ B_n \to S_n\ \underline{\text{od}})\delta = \nu G$,
   where $G = (\lambda f \in C_e \to_{dc} C_e \mid$
   $(\lambda q \in C_e \mid ([\bigvee i \mid 1\le i\le n \mid B_i] \vee q)$
   $\wedge [\bigwedge i \mid 1\le i\le n \mid B_i \Rightarrow (wlp'_e(S_i)\delta)\ f\ q]))$
7. $wlp'_e(A(E;v))\delta = pwlp'_e(A)(\delta)(E,v)$
8. $wlp'_e(p(E;v))\delta = pwlp'_e(p)(\delta)(E,v)$
9. $wlp'_e(\ |[\ \underline{\text{var}}\ x{:}\ t \mid S\ ]|\ ) = (\lambda q \in C_e \mid (wlp'_{e_1}((x \leftarrow y)S)\delta)q)$,
   where $y \in \underline{Name}$ such that $\vdash$ new (y,e) ,
   $e_1 = \text{Ext}(e,[y,t]_D)$.

□

Definition {pwp',pwlp'}

For all e ∈ *Env* the functions pwp'_e and $pwlp'_e$

$$\in L(\text{Id } \langle Name\rangle) \cup L(\text{Abstr } \langle Penv, Types, Types\rangle) \to \Delta \to P_e$$

are defined as follows:

Let $(\underline{\text{con}}\ x\colon t_1;\ \underline{\text{res}}\ y\colon t_2 \mid S) \in L(\text{Abstr}\ \langle \mathit{Penv}, \mathit{Types}, \mathit{Types}\rangle)$,

$E = L(\text{Expr}\ \langle e, \mathit{Prio}, t_1, \mathit{Nst}\rangle)$,

$V = L(\text{Var}\ \langle e, \mathit{Name}, t_2\rangle)$,

$x', y' \in \underline{\mathit{Name}}$ such that $\vdash$ new (x', e), $\vdash$ new (y', e), $x' \neq y'$,

$e' = \text{Ext}(e, [x', t_1]_D \cup [y', t_2]_D)$,

$S' = (x, y \leftarrow x', y')S$.

1.1. $pwp'_e((\underline{\text{con}}\ x\colon t_1;\ \underline{\text{res}}\ y\colon t_2 \mid S))\delta =$

$= (\lambda E \in E,\ v \in V \mid (x' \leftarrow E) \circ wp'_{e'}(S')\delta \circ (v \leftarrow y'))$.

1.2. $pwp'_e(p)\delta = \delta(p)(e)$.

2.1. $pwlp'_e((\underline{\text{con}}\ x\colon t_1;\ \underline{\text{res}}\ y\colon t_2 \mid S))\delta =$

$= (\lambda E \in E,\ v \in V \mid (x' \leftarrow E) \circ wlp'_{e'}(S')\delta \circ (v \leftarrow y'))$.

2.2. $pwlp'_e(p)\delta = \delta(p)(e)$.

□

Definition {wp and wlp for programs}

1.1. $wp(B) = wp'_e(B)\delta$,

where $e = \text{Empty}$, $\delta = \emptyset$.

1.2. $wp(p_1 = A_1, \ldots, p_k = A_k \mid B) = wp'_e(B)\delta$,

where $e = \text{Empty}$,

δ is the function $\{(p_1, \varphi_1), \ldots, (p_k, \varphi_k)\}$,

$(\varphi_1, \ldots, \varphi_k) = \mu(\Phi_1, \ldots, \Phi_k)$,

and, for i: $1 \leq i \leq k$:

$\Phi_i = (\lambda \varphi'_1, \ldots, \varphi'_k \in X \mid$

$(\lambda e \in \underline{\mathit{Env}} \mid pwp'_e(A_i)\{(p_1, \varphi'_1), \ldots, (p_k, \varphi'_k)\}))$.

2.1. $wlp(B) = wlp'_e(B)\delta$,

where e = Empty, $\delta = \emptyset$.

2.2. $wlp(p_1{=}A_1,\ldots,p_k{=}A_k \mid B) = wlp'_e(B)\delta$,

where e = Empty

δ is the function $\{(p_1,\psi_1),\ldots,(p_k,\psi_k)\}$

$(\psi_1,\ldots,\psi_k) = \nu(\Psi_1,\ldots,\Psi_k)$,

and, for i: $1 \leq i \leq k$:

$\Psi_i = (\lambda\psi'_1,\ldots,\psi'_k \in X \mid$

$(\lambda e \in \underline{Env} \mid pwlp'_e(A_i)\{(p_1,\psi'_1),\ldots,(p_k,\psi'_k)\}))$.

□

INDEX OF DEFINITIONS

{Unless indicated otherwise, numbers refer to numbers of definitions.}

{notions pertaining to grammars}

{notions pertaining to lattices}

{notions pertaining to condition transformers}

REFERENCES

[Apt 1]

Apt, K.R., Ten years of Hoare's logic: A survey - Part I. ACM Transactions on Programming Languages and Systems 3, 4 (Oct. 1981), pp. 431-483.

[Apt 2]

Apt, K.R., A sound and complete Hoare-like system for a fragment of Pascal, Report IW 97/78 Mathematisch Centrum, Amsterdam, 1978.

[Ashcroft]

Ashcroft, E.A., Wadge, W.W., R/ for Semantics, ACM Transactions on Programming Languages and Systems 4, 2 (Apr. 1982), pp. 283-294.

[Back 1]

Back, R.J.R., Proving total correctness of nondeterministic programs in infinitary logic, Acta Inf. 15 (1981), pp. 233-249.

[Back 2]

Back, R.J.R., Correctness preserving program refinements: proof theory and applications, Mathematical Centre Tracts 131, Mathematisch Centrum, Amsterdam, 1980.

[de Bakker]

de Bakker, J.W., Mathematical theory of program correctness, Prentice-Hall, 1980.

[Bjørner]

Bjørner, D., Jones, C.B. (eds.), The Vienna development method: the meta language, Lect. Not. Comp. Sci. 61, Springer, 1978.

[Bochmann]

Bochmann, G.V., Ledgard, H.F., Marcotty, M., A sampler of formal definitions, ACM Computing Surveys 8, 2 (June 1976), pp. 191-276.

[Brinch Hansen]

Brinch Hansen, P., Concurrent Pascal Report, Information Science, California Institute of Technology, 1975.

[Cook 1]

Cook, S.A., The complexity of theorem-proving procedures. Procs. 3rd Ann. ACM Symp. on theory of computing, May 1971.

[Cook 2]

Cook, S.A., Soundness and completeness of an axiom system for program verification, SIAM J. on Computing 7, 1978, pp. 70-90.

[Curry]

Curry, H.B., Feys, R.
Combinatory logic, vol. I., North-Holland Publ. Comp., 1958.

[Dijkstra 1]

Dijkstra, E.W., Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM 18, 8 (Aug. 1975), pp. 453-457.

[Dijkstra 2]

Dijkstra, E.W., A discipline of programming, Prentice-Hall, 1976.

[Dijkstra 3]

Dijkstra, E.W., Lecture notes "Predicate transformers" (Draft), EWD 835, 1982.

[Floyd]

Floyd, R.W., Assigning meanings to programs, Proc. Symp. in Applied Mathematics 19, Mathematical Aspects of Computer Science, J.T. Schwartz (ed.). AMS 1967, pp. 19-32.

[Ginsburg]

Ginsburg, S., Rounds, E.M., Dynamic syntax specification using grammar forms, IEEE Trans. Soft. Eng., vol. SE-4, no. 1, Jan. 1978, pp. 44-55.

[Goguen]

Goguen, J.A., Thatcher, J.W., Wagner, E.G., An initial algebra approach to the specification, correctness, and implementation of abstract data types; in: Current Trends in Programming Methodology, 4, R.T. Yeh (ed.), Prentice-Hall, 1978, pp. 80-149.

[Guttag]

Guttag, J.V., The specification and application to programming of abstract data types, Ph.D. thesis, Comptr. Syst. Res. Group Tech. Rep. CSRG 47, Dept. Comptr. Sci., Univ. of Toronto, 1975.

[Hemerik]

Hemerik, C., Relaties tussen taaldefinitie en taalimplementatie (in Dutch), MC Syllabus 42, J.C. van Vliet (ed.), Mathematisch Centrum, Amsterdam, 1980, pp. 109-142.

[Hoare 1]

Hoare, C.A.R. An axiomatic basis for computer programming, Comm. ACM 12, 10 (Oct. 1969), pp. 576-580.

[Hoare 2]

Hoare, C.A.R., Wirth, N., An axiomatic definition of the programming language Pascal, Acta Inf. 2, 1973, pp. 335-355.

[Hoare 3]

Hoare, C.A.R., Procedures and parameters: an axiomatic approach; in: Symp. on Semantics of Algorithmic Languages, E. Engeler (ed.), Lect. Not. Math. 188, Springer, 1971, pp. 102-116.

[Jazayeri]

Jazayeri, M., Ogden, W.F., Rounds, W.C., The intrinsically exponentional complexity of the circularity problem for attribute grammars, Comm. ACM 18, 12 (Dec. 1975), pp. 697-706.

[Knuth]

Knuth, D.E., Semantics of context-free languages, Math. Syst. Theory 2 (1968), pp. 127-145. Correction in: Math. Syst. Theory 5 (1971), p. 95.

[Karp]

Karp, C.R., Languages with expressions of infinite length, North-Holland Publ. Comp., 1964.

[Lauer]

Lauer, P.E., Consistent formal theories of the semantics of programming languages, Tech. Rep. TR. 25.121, IBM Lab. Vienna, Nov. 1971.

[Ledgard]

Ledgard, H.F., Production systems: a notation for defining syntax and translation, IEEE Trans. Soft. Eng., vol. SE-3, no. 2, March 1977, pp. 105-124.

[Mc Carthy]

Mc Carthy, J., Towards a mathematical science of computation, Procs. IFIP Congress 62, C.M. Popplewell (ed.), North-Holland 1963, pp. 21-28.

[Milne]

Milne, R.E., Tranforming predicate transformers, Proc. IFIP TC-2 Working Conference on Formal Description of Programming Concepts, E.J. Neuhold (ed.), North-Holland, 1978, pp. 31-65.

[Plotkin]

Plotkin, G.D., Dijkstra's predicate transformers and Smyth's power domains, Procs. 1979 Copenhagen Winter School, D. Bjørner (ed.), Lect. Not. Comp. Sci. 86, Springer, 1980.

[Räihä]

Räihä, K.-J., Bibliography on attribute grammars, ACM SIGPLAN Notices 15, 3 (March 1980), pp. 35-44.

[Scott 1]

Scott, D.S., Logic with denumerably long formulas and finite strings of quantifiers, Symp. on the Theory of Models, J. Addison, L. Henkin, A. Tarski (eds.), North-Holland, 1965, pp. 329-341.

[Scott 2]

Scott, D.S., Data types as lattices, SIAM J. on Computing 5, 1976, pp. 522-587.

[Simonet]

Simonet, M., An attribute description of a subset of ALGOL 68, Proc. of the Strathclyde ALGOL 68 Conf., ACM SIGPLAN Notices 12, 6 (June 1977), pp. 129-137.

[Strachey]

Strachey, C., Wadsworth, C.P., Continuations: a mathematical semantics for handling full jumps, Technical Monograph PRG-11, Programming Research Group, University of Oxford, 1974.

[Stoy]

Stoy, J., Denotational semantics: the Scott-Strachey approach to programming language theory, MIT Press, 1977.

[Tennent]

Tennent, R.D., Language design methods based on semantic principles, Acta Inf. 8 (1977), pp. 97-112.

[Wand]

Wand, M., A characterization of weakest preconditions, J. Comp. Syst. Sci. 15, 1977, pp. 209-212.

[Watt]

Watt, D.A., An extended attribute grammar for Pascal, SIGPLAN Notices 14, 2 (Feb. 1979), pp. 60-74.

[Wegner]

Wegner, P., The Vienna Definition Language, Computing Surveys 4, 1972, pp. 5-63.

[Wirth 1]

Wirth, N., Weber, H., EULER: A Generalization of ALGOL, and its Formal Definition, part II, Comm. ACM 9, 2 (Feb. 1966), pp. 89-99.

[Wirth 2]

Wirth, N., Hoare, C.A.R., A contribution to the development of ALGOL, Comm. ACM 9, 6 (June 1966), pp. 413-431.

[van Wijngaarden]

van Wijngaarden, A. et. al. (eds.), Revised report on the algorithmic language ALGOL 68, Acta Inf. 5, 1975, pp. 1-236.

SAMENVATTING

Het onderzoek waarvan in dit proefschrift verslag wordt gedaan maakt deel uit van een meer omvattend project dat tot doel heeft het op systematische wijze construeren van correcte implementaties van programmeertalen. De volgende aspecten spelen daarbij een rol:

1. De definitie van de brontaal.
2. De definitie van de doeltaal.
3. De constructie van een "betekenis behoudende" afbeelding van brontaal naar doeltaal.
4. De constructie van een programma dat die afbeelding realiseert.

Het is duidelijk welke afhankelijkheden er tussen deze aspecten bestaan: 3 is uitsluitend afhankelijk van 1 en 2, en de specificaties van 4 zijn gebaseerd op 1 en 3. Het is ook duidelijk dat de correctheidsoverwegingen van 3 en 4 gescheiden kunnen worden en dat de betrouwbaarheid van de resulterende vertaler uiteindelijk bepaald wordt door de mate van precisie, volledigheid en ondubbelzinnigheid van 1 en 2.

Het onderwerp van dit proefschrift is het ontwerp en de formele definitie van een brontaal SL en een doeltaal TL, die dienen als uitgangspunt voor een implementatieproces zoals boven geschetst. Bovendien worden de daarvoor benodigde definitiemethoden zover als nodig ontwikkeld. Gezien de gegeven achtergrond zal het echter duidelijk zijn dat dit proefschrift niet beschouwd dient te worden als een op zichzelf staande studie van taaldefinitie. Het grootste deel van het beschreven werk is bedoeld als theoretische fundering van het genoemde implementatieproces.

In het proefschrift wordt allereerst kort ingegaan op voorwaarden waaraan formele taaldefinities dienen te voldoen, zoals beschikbaarheid van goede wiskundige theorie en afwezigheid van overspecificatie en implementatieaspecten. Vervolgens wordt in hoofdstuk 2 de syntactische definitie van de brontaal behandeld. Het belangrijkste onderwerp van dit hoofdstuk is de ontwikkeling van een variant van de welbekende attribuutgrammatica's [Knuth], die primair gericht is op taalspecificatie. De belangrijkste

componenten van deze variant zijn een verzameling geparameteriseerde produktieregels en een zogenaamde attribuutstructuur, met behulp waarvan eigenschappen van parameters uit gegeven axioma's afgeleid kunnen worden. Enerzijds kan een attribuutgrammatica van deze soort beschouwd worden als een zuiver formeel systeem gebaseerd op herschrijfregels en logische afleidingen. Anderzijds kan de attribuutstructuur, die overeenkomt met een algebraïsche type-specificatie in de zin van [Goguen, Guttag], direct gebruikt worden als specificatie van dat deel van een vertaalprogramma, dat de context-afhankelijke analyse uitvoert.

In hoofdstuk 3 wordt de basis gelegd voor de semantische definitie van bron- en doeltaal. De gebruikte definitiemethode is in essentie die van de "predicate transformers" [Dijkstra 1, Dijkstra 2]. Deze methode wordt eerst gefundeerd met behulp van een variant van Scott's "theory of continuous lattices" [Scott 2] en "infinitary logic" [Back 1, Karp]. Daarna worden de predicate transformers voor een deel van de brontaal in dit raamwerk beschouwd. Tenslotte worden deze resultaten gebruikt om logische systemen in de stijl van [Hoare 1, Hoare 2] te ontwikkelen voor het bewijzen van partiële en totale correctheid.

In hoofdstuk 4 worden de methoden van hoofdstukken 2 en 3 toegepast op andere constructies van de brontaal, te weten blokken en procedures. Voor deze constructies worden syntaxis, semantiek en bewijsregels ontwikkeld. De verschillende aspecten van procedures worden zoveel mogelijk in isolement behandeld. De behandeling van blokken in sectie 4.1 dient voornamelijk om de effecten van het introduceren van lokale namen te onderzoeken. In sectie 4.2 worden aan de hand van zogenaamde abstracties de gevolgen van parameterisering bestudeerd. In sectie 4.3 wordt met behulp van de lattice theory van sectie 3.1 een betrekkelijk eenvoudige behandeling van parameterloze recursie gegeven. Tenslotte worden in sectie 4.4 de verscheidene aspecten samengevoegd, hetgeen resulteert in een behandeling van recursieve procedures met parameters.

In hoofdstuk 5 worden enige aspecten van de formele definitie van de doeltaal behandeld, met name die welke betrekking hebben op instructies die de volgorde van verwerking beïnvloeden, zoals sprongen en subroutine-aanroepen. Het doel van dit werk is het ontwikkelen van predicate transformers voor machine-instructies, die vervolgens gebruikt kunnen worden bij het construc-

eren van correcte implementaties. In eerste instantie wordt met behulp van de lattice theory uit sectie 2.1 en de techniek van de "continuations" [Strachey] een predicate transformer semantiek ontwikkeld. Vervolgens wordt uit deze definitie via enige transformaties een equivalente operationele beschrijving door middel van een interpretator-programma afgeleid. Deze afleiding is zowel een bewijs van de consistentie van twee definities als een voorbeeld van het afleiden van een implementatie uit een niet-operationele definitie. Bovendien geeft deze afleiding ook een indruk van de semantiek behoudende transformaties die bij de vertaling van brontaal naar doeltaal een rol zullen spelen.

Hoofdstuk 6 bevat een korte nabeschouwing van het werk.
Appendix A bevat bewijzen van enige lemma's.
Appendix B bevat de verzamelde definitie van de brontaal.

CURRICULUM VITAE

De schrijver van dit proefschrift werd op 24 april 1952 geboren te Leiden. In 1969 behaalde hij aan de Mathenesser H.B.S. te Rotterdam het diploma H.B.S.-B. Hij studeerde vervolgens wiskunde aan de Technische Hogeschool Delft, alwaar hij in november 1976 het ingenieursexamen aflegde. Afstudeerhoogleraar was prof.dr.ir. W.L. van der Poel. Sinds 1 april 1977 is hij werkzaam bij de Technische Hogeschool Eindhoven, meer in het bijzonder in de Onderafdeling der Wiskunde en Informatica bij prof.dr. F.E.J. Kruseman Aretz.

STELLINGEN

behorende bij het proefschrift

Formal definitions of programming languages as
a basis for compiler construction

van

C. Hemerik

Eindhoven,
15 mei 1984.

STELLINGEN

1. Elke recursief opsombare taal kan gedefiniëerd worden met een attribuutgrammatica zoals gedefiniëerd in hoofdstuk 2 van dit proefschrift.

2. De in de literatuur overheersende opvatting, dat een operationele definitie van een programmeertaal het beste uitgangspunt vormt voor implementatie van die taal, is onjuist.

3. Programmeertalen zijn artefacten. Derhalve verdient het ontwerpen van talen met wenselijke eigenschappen meer aandacht dan het bestuderen van bestaande talen.

4. Het is goed te bedenken, dat een aantal belangrijke doorbraken in de informatica het gevolg zijn van het zorgvuldig beperken van de combinatorische vrijheid die geboden wordt door het Von Neumann-berekeningsmodel.

5. Het vak programmeren is de laatste vijftien jaar onmiskenbaar wiskundiger van aard geworden. De verworven inzichten zijn echter nog onvoldoende in de vorm van stellingen vastgelegd.

6. Informatica is bij uitstek een ingenieurswetenschap.

7. Het aantrekken van grote aantallen informaticastudenten schaadt de kwaliteit van onderwijs en onderzoek in de informatica.

8. Een informatica-ingenieur dient een zekere rijpheid te bezitten om de ontwikkelingen in zijn vakgebied kritisch te kunnen beschouwen. Een eerste - fase opleiding van slechts vier jaar biedt voor het benodigde rijpingsproces onvoldoende ruimte.

9. De veelgenoemde achterstand van Nederland op het gebied van de informatica is slechts vermeend.

10. Gezien de omstandigheden waaronder proefschriften hun voltooiing naderen, verdient het aanbeveling te onderzoeken of er verband bestaat tussen de wet van Parkinson en de paradox van Zeno.