University of Southampton

Program Refinement using a Universal Law: Language Specification and Prototype Tool

David William Roff Marsh

Thesis submitted in candidature for the
degree of Doctor of Philosophy

Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

September 1999
(including corrections, April 2000)

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

Program Refinement using a Universal Law: Language Specification and Prototype Tool.

David William Roff Marsh

The refinement calculus introduced by Back [Bac78], Morgan [Mor88a], Morris [Mor87] and others, combines specifications and programs into a single language and allows the development of a program from its specification to proceed in a number of small steps. The subject of this thesis is the definition of a specification and programming language based on the refinement calculus. A prototype tool has been developed from a formal language definition and simple examples are used to evaluate the language and tool.

One difference from existing refinement systems is the use of a universal refinement law and a simple compiler-style tool, rather than many laws and interactive tools. Most of the benefits of the refinement calculus are maintained while the change of programming method is smaller than that required to use an interactive refinement system. The size of refinement steps can be varied and complex derivations of obvious developments avoided. Context is propagated automatically, allowing Back's generalisation of the familiar assignment statement to be used for specification.

The use of subtypes and dependent types is another feature distinguishing this work. Subtypes and dependent types are integrated smoothly into the programming language and are shown to provide an accurate model of the runtime constraints of a programming language.

Some have argued that formal methods are necessary to achieve safe programmable systems, but to date this has not been borne out by experience. However, achieving dependable software is expensive, suggesting that the use of formal methods should be directed towards reducing costs. This vision can be realised only by exploiting theories such as the refinement calculus pragmatically, aiming to enhance existing practices, with flexibility in matters such as the degree of rigour and the order of development steps. The work described in this thesis is intended to be a step in this direction.

Acknowledgements

I am grateful to my supervisor, Andy Gravell, for his support and encouragement thoughout my long period of study. The patience and interest of colleagues past and present at Program Validation and ERA Technology is also acknowledged.

My greatest debt is to my wife Julia, without whose support this thesis would never have been completed, and to Elizabeth and Clare for their unwitting contribution.

Contents

1 Introduction **1**

1.1 Relationship to Existing Work 1

1.2 Designing a Language of Refinement 5

1.3 Prototype Tool Support 6

1.4 Thesis Outline 6

I Overview and Application **8**

2 Foundations **9**

2.1 The λ-Cube 9

2.2 Predicate Transformers and Refinement 13

3 Language Overview **19**

3.1 Terms and Types 20

3.2 Declarations 27

3.3 Datatype Declarations and Terms 29

3.4 Standard Declarations 32

3.5 Statements 36

3.6 Units 42

4 Case Studies **46**

4.1 Largest Rectangle under a Histogram 46

4.2 Integers of Finite Range 53

4.3 Graph Sink 60

5 Prototype Tool Support **70**

5.1 Objectives 70

5.2 Design 71

5.3 Simplication of Proof Obligations 76

6 Discussion **78**

6.1 Verification and Refinement Systems 78

6.2 Languages with Dependent Types 85

6.3 Comparison and Evaluation 90

II Language Specification **106**

7 Overview of the Language Specification **107**

7.1 Objectives of the Language Specification 107

7.2 Language Components 107

7.3 Structure of the Specification 108

7.4 Notation 109

8 Environments, Declarations and Theories **110**

8.1 Environments 110

8.2 Simple Declarations 112

8.3 Theories 115

9 Terms **117**

9.1 Using the λ-Cube 117

9.2 Abstract Syntax 121

9.3 Visibility of Identifiers 122

9.4 A Type Checking Algorithm 123

9.5 Type Errors 130

9.6 Inference of Implicit Arguments 132

10 Datatypes and Recursion **136**

10.1 Datatype Declarations 136

10.2 Case Terms 140

10.3 Primitive Recursion 144

10.4 General Recursion 149

11 Statements **153**

11.1 Abstract Syntax 153

11.2 Visibility of Identifiers 155

11.3 Statement Well-formation 160

11.4 Well-formation of Assignment Statements 163

11.5 Procedure Statements Well-formation 168

11.6 Semantics of Statements 177

11.7 Semantics of Procedure Statements 183

11.8 Semantics of the Loop Statement 185

11.9 Semantics of Recursion 189

11.10 Context Propagation 192

12 Refinement Texts **195**

12.1 The Refinement Statement 195

12.2 Revisiting the Recursion Statement 198

12.3 Refinement Texts 199

12.4 Validity of Checking Refinements 200

13 Conclusions **201**

13.1 Summary of Achievements 201

13.2 Further Work 202

13.3 Refinement in the Mainstream 203

A Syntax Definition 204

B Standard Declarations 213

C Proofs 220

Chapter 1

Introduction

The refinement calculus introduced by Back [Bac78], Morgan [Mor88a], Morris [Mor87] and others, extends the theory of program correctness introduced by Hoare [Hoa69] and Dijkstra [Dij75, Dij76]. Its principle advantage is to combine specifications and programs into a single language and allow the development of a program from its specification to proceed in a number of small steps, each of which can be justified independently of the others.

The theory of refinement and its notation are powerful intellectual tools which can be used, with more or less formality, to develop real programs. A number of examples of development using refinement have been published [Abr91a, But97, CR91, Gra91, Mor94, Woo91, SS99]. However, the use of refinement techniques for problems of any significant size requires *notation* to be replaced by *precisely-defined language*, supported by computerised tools.

The subject of this thesis is the design of such a language, based on the refinement calculus. The well-formation rules and proof semantics of the language are formally specified and a prototype tool developed from the specification. Some simple examples are used to evaluate the language and tool.

1.1 Relationship to Existing Work

The language design presented in this thesis differs from existing refinement languages and tools in two main ways:

(i) a universal refinement law is used to demonstrate the correctness of refinement steps,

(ii) the language is built on a type system that includes subtypes.

We introduce these key aspects of our approach below.

1.1.1 Using a Universal Refinement Law

Existing tools supporting the refinement calculus, which are surveyed in Chapter 6, support interactive transformation of a specification to an implementation. The transformation is carried out using *refinement laws*, which are proved from the semantics of the language. Some refinement laws always apply, whilst others have side-conditions which may require proof in predicate logic. We show that the refinement calculus can be applied non-interactively using a 'universal' refinement law, with program construction separated from correctness proof. The advantages of this approach are that:

(i) the existing programming method is maintained,

(ii) refinements can be made in steps of any size,

(iii) verification is separated from programming, allowing proof to be delayed or other verification techniques to be employed, and

(iv) development tools are similar to existing tools.

Programming Method It is a major change to the programming method to make programming nothing more than the successive application of refinement laws. Although the identification of refinement laws (such as the collection in [Mor94]) or, more generally, a programming language algebra [HHJ+87], provides the programmer with a new understanding of the meaning and correctness of programs, it is not necessary to make such a major change to the programming method to benefit from the refinement calculus. Instead, refinement should offer a new approach to the programmer which builds on existing practices.

Refinement in Flexibly-Sized Steps Interactive refinement using refinement laws proceeds in small steps. There is potential for extending the set of refinement laws and for composing laws into refinement tactics, but this is not well developed. At the other extreme, an earlier generation of programming verification systems, such as Gypsy [Amb77], required each procedure to be verified as a whole.

We show that the refinement calculus can be used in the manner of the system such as Gypsy, but with support for refinement steps of variable size. The user can choose either the small steps of the refinement calculus or larger steps. The latter may be appropriate when the circumstances demand a less detailed justification of correctness or when the overall refinement is more obvious than the detailed derivation.

Verifying Refinement Steps The construction of the refinement text is itself the main component of the correctness argument for the program. In many applications, the cost of formally proving the lemmas of this argument – the verification conditions

– will not be justified. Instead, further verification can be achieved by inspection, or by other means short of formal proof. This topic is beyond the scope of this thesis, but we review the prospect of integrating refinement with other verification techniques in the final chapter.

Systems such as Gypsy are sometimes derided as requiring *posit and verify*, meaning that formal verification is attempted on a program when there is no basis for confidence that the program is correct. In our approach, stepwise development is the primary means of ensuring correctness[1].

A possible disadvantage of not using laws of refinement is that more correctness proofs will be required, repeating the justification for the development step captured in the refinement law. However, many laws have side conditions showing that the law is applicable. Using simple examples, we compare the proof obligations arising from our approach with the side conditions of refinement laws.

Tool Support Existing tools supporting the refinement calculus are integrated with theorem provers, leading to a tool architecture very different from mainstream programming language tools. In particular, the representation of the programming language is internal to the tool, proof may subsume other forms of static error checking and the order of development is constrained.

We show that the refinement calculus can be used with a conventional ASCII language syntax and a support tool which resembles a compiler, with well-formation checking separate from proof. This has the advantage of simplicity. In addition, we believe that it provides the best approach for future support of the refinement calculus in an integrated development environment, similar to those available for existing programming languages.

1.1.2 Specification and Program Types

The language of refinement must express both specifications and programs. To be useful as a specification language, the expressions and types of the language must be much more expressive than those included in a programming language such as Pascal. Sets, functions and free datatypes (such as lists) are required, with conditional and recursive definitions as well as simple definition by equality.

Subtypes and Dependent Types We adopt a language with subtypes and dependent types, based on higher order functions. The language draws particularly on the

[1] *Posit and verify* is really a characterisation of way a method is used rather than of the method itself. In the author's experience using SPADE [COCD86], a system similar to Gypsy, care was taken to ensure correctness before doing formal verification, e.g. by writing the program to make correctness as obvious as possible.

Veritas$^+$ [HDL90a] [HDL90b] and PVS [ORS92] systems, both of which are interactive theorem provers.

The inclusion of subtypes means that type checking is not decidable, since a proof may be needed to show that a term belongs to a subtype. In Veritas$^+$ construction and type checking of terms are carried out interactively. This approach allows a user to prove subtype membership when required, while simpler cases can be automated using tactics. Here, we follow the approach of PVS, where type checking is non-interactive, since this is the long established approach used for programming languages. The type checker reports obvious type errors and, if necessary, produces proof obligations, called *type correctness conditions*. These must be proved to show that the specification is type correct.

One approach considered was to adopt the language of PVS in the language of refinement, without modification. However, the PVS type checker is embedded in the PVS prover, so to use the PVS language directly requires the refinement tool to be based on the PVS prover or the type checker to be re-implemented. Instead, drawing on the ideas of both PVS and Veritas$^+$, a simple language with subtypes and dependent types has been designed. The main argument of this thesis concerns the suitability of a higher-order logic with subtypes and dependent types for use within a language of refinement. However, during the design some differences have emerged between our language and that of PVS, which provided our starting point.

Typed Specification Languages The desirability of having a typed specification language has been disputed, for example by Lamport and Paulson [LP98], who argue that types are not necessary in specification languages and that type systems reduce flexibility and add complexity. However, popular specification languages such as Z [Spi89] and VDM [Jon86] are typed and types allow simple errors to be detected automatically. For this reason, we adopt a typed specification language, despite the disadvantages identified by Lamport and Paulson. Some points raised by Lamport and Paulson concerning dependent types are considered in Chapter 6.

Types and Proof Introducing the PVS theorem prover [ORSH95], Rushby and others argue that logical languages allowing only total functions are more amenable to efficient automatic theorem proving than those, such as Z, which allow partial functions. Further evidence for this view comes from Martin [Mar97] who cites partial functions as one of the reasons why effective tool support for Z proof is hard[2].

The PVS developers show how the use of subtypes allows functions which would normally be considered to be partial to be made total functions. In this approach, the division function $_/_$ is total, but the domain of the divisor is non-zero numbers; the

[2]Saaltink [Saa97] describes how Z/EVES handles partial functions (and other causes of undefinedness in Z) and reports that many published Z specification contain errors caused by misuse of partial functions.

expression $1/0$ is not well typed. In set theory or in higher order logic without subtypes, $1/0$ is an arbitrary value[3].

Programming Language Types Lamport and Paulson consider only specification languages, noting that "the advantages of typed programming languages are obvious". A language of refinement is both a specification and programming language, suggesting the use of types. Subtypes and partial functions are common in programming languages. Examples include integers of limited range, which are a subtype of mathematical integers, and partial functions such as arrays and collections (indexed by pointers).

One alternative to subtypes is to add 'type declarations' as part of the derivation of a program from a specification, for example using invariants, as described by Morgan [Mor89]. The technique of totalising partial functions with arbitrary values, another alternative, is problematic in a *programming* language, where application of an argument outside the domain of a partial function may lead to a run-time error. These alternatives are reviewed in more detail in Chapter 6.

1.2 Designing a Language of Refinement

Our language is based on the extensions to Dijkstra's guarded command language introduced by Morgan and Back and on existing languages of subtypes and dependent types. For the syntax, we follow the example of existing languages, particularly Pascal and its descendants, avoiding unnecessary innovation wherever possible. But there are novel issues to be considered in the design of a language of refinement. We summarise our approach to two of these below.

Factorisation An advance in the theory of predicate transformers since Dijkstra's work has been to identify the essential statement forms from which more complex forms can be composed. One example of this is the separation of procedure definition from parameterisation [Mor88c]. As well as simplifying the theory, aspects of this factorisation can be made apparent to the user of the language; this approach is followed in our treatment of both recursion and parameterisation.

The very simple command language, which is shown to be complete by Back and von Wright [BW89a], is a more radical factorisation. The statements available to the programmer are defined from the primitive commands. This factorisation would be especially attractive if the generation of correctness conditions could be carried out by expanding the statement definitions. However, the encoding of the assignment statement using the primitive commands is complex and this is likely to lead to complex

[3]An arbitrary number in a typed logic, whereas in un-typed set theory it could be any value.

correctness conditions[4]. We prefer to specify the meaning of the statements used by the programmer directly.

Refinement Syntax In a refinement language, a tree-like structure replaces the linear text of a conventional programming language. Some syntax is required to represent this refinement tree. Although it is clear that interactive tools, such as folding editors, will be useful, we consider that the representation of the tree should not be private to such tools. Instead, we adopt a conventional ASCII syntax for the refinement tree. Other approaches include Back's refinement diagrams [Bac91] or the statement marking convention used by Morgan [Mor94].

1.3 Prototype Tool Support

The well-formation checking and refinement verification condition generator have been implemented in a prototype tool, aiming to follow the formal specification as closely as possible. To achieve this, the lazy functional language Haskell [PE97] was chosen for the implementation. The prototype implementation serves two purposes:

(i) it allows the formal specification to be validated by testing, and

(ii) it allows the difficulty of discharging the proof obligations for both type correctness and refinement to be investigated.

1.4 Thesis Outline

The first part of the thesis introduces the language and describes its application. Chapter 2 reviews the theoretical foundations on which the language is built: the typed λ-calculus and the refinement of predicate transformers. The specification and refinement language is introduced in Chapter 3, with a rationale for the major design decisions. Example specifications and refinements are presented in Chapter 4 with an examination of the proof obligations generated by the prototype tool. The prototype tool is described in Chapter 5. Existing work is surveyed in Chapter 6, including languages based on subtypes and dependent types and existing tool support for the refinement calculus. This is followed by an evaluation of our work in comparison with existing work.

The specification of the program refinement language appears in Part II. Chapter 7 introduces the specification, in particular the use of de-Bruijn indices as a mechanism

[4] Abrial [Abr89] has successfully followed a similar approach, using *generalised substitutions* which make assignment a primitive statement.

for specifying the scope rules of the language. Subsequent chapters address each part of the languages: declarations and theories are covered in Chapter 8, terms in Chapters 9 and 10, statements in Chapter 11 and finally refinement texts in Chapter 12.

Overall conclusions appear in Chapter 13.

There are three appendices: the language syntax definition is in Appendix A and Appendix B contains standard declarations. Proofs relating to the properties of statements are in Appendix C.

Part I

Overview and Application

Chapter 2

Foundations

In this chapter the two foundations of the language and tool are reviewed: the typed λ-calculus and program refinement.

2.1 The λ-Cube

The starting point for the specification language is the typed λ-calculus λC. λC is one of eight systems of typed λ-calculus described in a uniform manner by the λ-cube [Bar92]; this section is based on Barendregt's description. λC is similar to the *Calculus of Constructions* [CH88].

2.1.1 Terms and Types

In λC, term and types are not distinguished syntactically. The following abstract syntax defines a set of *pseudo-terms* $\mathcal{T}$:

$$\mathcal{T} = V \mid C \mid \mathcal{T}\ \mathcal{T} \mid \lambda V : \mathcal{T}.\mathcal{T} \mid \Pi V : \mathcal{T}.\mathcal{T}$$

where V and C are infinite collections of variables and constants respectively.

Informally, the pseudo-term $\lambda x : A.B$ denotes a function, with the type of the bound variable and therefore the domain type of the function, made explicit. The pseudo-term $\Pi x : A.B$ is dependent version of the function type $A \to B$, in which the range type may depend on the element of the domain type. When the bound variable x is not free in B, we use $A \to B$ as an abbreviation for $\Pi x : A.B$.

2.1.2 Reduction

On $\mathcal{T}$, the notion of β-reduction is defined by the following contraction rule:

$$(\lambda x : A.B)\ C \rightarrow_\beta B[x := C]$$

where $B[x := C]$ denotes the syntactic substitution of C for the variable x in B, renaming bound variables in B where necessary to avoid capturing any free variables in C. The relation $\twoheadrightarrow_\beta$ is the reflexive transitive closure of $\rightarrow_\beta$. The congruence relation between terms $A =_\beta B$, with terminology A is β-convertible to B, is defined from β-reduction in the standard way.

2.1.3 Type Assignment

Statements A *declaration* is of the form $x : A$ with $A \in \mathcal{T}$ and x a variable. A *statement* is of the form $A : B$ with $A, B \in \mathcal{T}$; here, A is the *subject* of the statement and B its *predicate*. A *context* is a finite ordered sequence of declarations with distinct subjects.

If	Γ	denotes	$< x_1 : A_1, \ldots, x_n : A_n >$
then	$\Gamma, y : B$	denotes	$< x_1 : A_1, \ldots, x_n : A_n, y : B >$

The empty context may be written $<>$.

Sorts Two constants are selected and given the names $*$ and $\Box$; these constants are called *sorts*. Let $\mathcal{S} = \{*, \Box\}$ and let s range over $\mathcal{S}$. Some examples of possible derivations are:

$* : \Box$	types
$* \rightarrow * : \Box$	unary function types
$* \rightarrow * \rightarrow * : \Box$	binary function types
...	

We refer to the elements of $\Box$ (that is, terms K for which $K : \Box$ can be derived) as *kinds*, and elements of $*$ as *types*.

Type assignment The type assignment rules axiomatize the notion

$$\Gamma \vdash A : B$$

stating that the statement $A : B$ can be derived in the context Γ. Figure 2.1 shows the type assignment rules. $A, B, F, a, b, \ldots$ are arbitrary pseudo-terms and x is an arbitrary variables. The different systems on the λ-cube are distinguished by the range of s_1 and s_2 in the product rule. For λC,

$$(s_1, s_2) \in \{(*, *), (\Box, *), (*, \Box), (\Box, \Box)\}$$

(axiom)	$<>\vdash * : \Box$
(start rule)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$
(weakening rule)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$
(application rule)	$\dfrac{\Gamma \vdash F : (\Pi x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\ a : B[x := a]}$
(abstraction rule)	$\dfrac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}$
(conversion rule)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}\ B =_\beta B'$
(product rule)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_2}$

Figure 2.1: Type assignment for λC

2.1.4 Classifying Terms

A map $\sharp : \mathcal{T} \to \{0, 1, 2, 3\}$ can be defined [Bar92, Definition 5.2.23], with the following properties:

$$\sharp\,(\Box) = 3$$
$$\Gamma \vdash A : B \Rightarrow \sharp(A) + 1 = \sharp(B)$$

This provides a simple classification of legal terms, for which we use the following terminology:

$\sharp(A) = 0$	A is an element
$\sharp(A) = 1$	A is a type
$\sharp(A) = 2$	A is a kind

One corollary of this property is that the term $\Box$ can never be the subject of a statement.

It is convenient to use the terms 'type' and 'element' in a more general sense: when $a : A$ is a statement, we call A the type of a and a an element of A. This corresponds with the classification above only when $\sharp(a) = 0$.

2.1.5 Properties of λC

It this section we quote a number of properties of λC. The following terminology is used: a context Γ is called *legal*, if there exists $A, B \in \mathcal{T}$ such that $\Gamma \vdash A : B$. In a legal

context Γ, A is called a (legal) Γ-*term* if, for some $B \in \mathcal{T}$: then $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$. Let A, B, C, D be pseudo-terms, and let Γ, Δ be contexts.

Substitution ([Bar92, Lemma 5.2.11]) A variable can be replaced by a term of the same type. If $\Gamma, x : A, \Delta \vdash B : C$ and $\Gamma \vdash D : A$ then $\Gamma, \Delta[x := D] \vdash B[x := D] : C[x := D]$.

Classification ([Bar92, Corollary 5.2.14(1)]) If $\Gamma \vdash A : B$ then, for some $s \in \mathcal{S}$, either $B \equiv s$ or $\Gamma \vdash B : s$.

Subterm legality ([Bar92, Corollary 5.2.14(4)]) Any subterm of a legal term is also a legal term.

Subject and predicate reduction ([Bar92, Theorem 5.2.15]) β-reduction can be used to change the subject of a statement. If $\Gamma \vdash A : B$ and $A \twoheadrightarrow_\beta A'$ then $\Gamma \vdash A' : B$. The predicate of a statement can be replaced similarly.

Uniqueness of types ([Bar92, Lemma 5.2.21]) Types are unique upto β-conversion. If $\Gamma \vdash A : B_1$ and $\Gamma \vdash A : B_2$ then $B_1 =_\beta B_2$.

Strong Normalisation ([Bar92, Lemma 5.3.3]) Legal terms in λC are strongly normalising, that is, there are no infinite sequences of β-reductions.

Decidability of type checking and typability ([Bar92, Lemma 5.2.18]) Here type checking means deciding the truth or falsity of the statement: $\Gamma \vdash A : B$ for given Γ, A and B. In a given context Γ a term A is typable if there exists a term B such that: $\Gamma \vdash A : B$.

Uninhabited type ([Bar92, Proposition 5.2.31]) The type $(\Pi\alpha : *.\alpha)$ is not inhabited in λC. The proof of this proposition shows that a term of this type would not have a normal form.

2.1.6 Logic

Type systems such as λC have been used to represent logic by interpreting propositions as types. The approach, attributed to N.G. de Bruijn, is described by Barendregt [Bar92, §5.1].

A system of logic can be represented by introducing a type *prop* which is assumed to be closed under implication.

$$prop : * \qquad \supset : prop \rightarrow prop \rightarrow prop$$

To this context, we add a variable $T : prop \rightarrow *$ with the interpretation that $\phi : prop$ is valid if $T\ \phi$ is inhabited, that is there exists a term M, such that $M : T\ \phi$. We can express the properties of implication by introducing functions to eliminate and introduce implication:

$$\supset_e\ \phi\ \psi : T(\phi \supset \psi) \rightarrow T\ \phi \rightarrow T\ \psi$$
$$\supset_i\ \phi\ \psi : (T\ \phi \rightarrow T\ \psi) \rightarrow T(\phi \supset \psi)$$

The same effect can be achieved by using $*$ for *prop*, making T the identity and using $\rightarrow$ for $\supset$.

The feasibility of building proof systems in a purely constructive manner has been shown by others, notably the Nuprl project [CAB+86]. A particular feature of this approach is that a proof can be interpreted as a program, so that the activities of writing a program and proving it correct are combined. However, this approach is still immature.

2.2 Predicate Transformers and Refinement

In this section, we summarise the theory of predicate transformers and refinement, which is used in the remainder of the thesis.

2.2.1 Refinement

If I, I' are implementations and *Spec* a specification, we write:

$$I \textbf{ sat } \mathit{Spec}$$

to show that I implements the specification *Spec* according to some satisfaction relation. Provided that this satisfaction relation considers all the properties of implementations which are of interest to us, another implementation I' will be an acceptable replacement for I, provided that I' satisfies any specification satisfied by I, In these circumstances, I' is called a refinement of I, and the refinement relation $\sqsubseteq$ is defined by:

$$I \sqsubseteq I' \mathrel{\widehat{=}} \forall \mathit{Spec} \cdot I \textbf{ sat } \mathit{Spec} \Rightarrow I' \textbf{ sat } \mathit{Spec}$$

This fundamental definition of refinement is used below to derive a refinement relation for the total correctness of sequential programs. In a wide spectrum language, where specifications are not distinguished from implementations, the **sat** and $\sqsubseteq$ relations also become the same. The $\sqsubseteq$ relation is both reflexive and transitive as an immediate consequence this definition.

Monotonicity Let J be an implementation, which is a component of some large implementation I. To indicate this relationship, we may write I as $I[J]$. So that the refinement relation can be used in the method of stepwise refinement, proposed by Wirth [Wir71], we require the property that refining a component of the implementation yields a refinement of the complete implementation.

$$J \sqsubseteq J' \Rightarrow I[J] \sqsubseteq I[J']$$

Let $\otimes$ be an arbitrary operator composing implementations into implementations (without loss of generality, we show $\otimes$ as a binary operator). Then stepwise refinement is

supported provided that the refinement relation is monotonic with respect to the arguments of $\otimes$, i.e. for any implementations I, I', J:

$$I \sqsubseteq I' \Rightarrow (I \otimes J) \sqsubseteq (I' \otimes J)$$

and

$$I \sqsubseteq I' \Rightarrow (J \otimes I) \sqsubseteq (J \otimes I')$$

2.2.2 Weakest Precondition of Statements

We introduce a language of statements, with statements acting as both specifications and implementations. Statements are identified with their weakest preconditions [Dij76], which gives the weakest predicate describing the initial state which will ensure that the statement terminates in the final state described by a given postcondition. Despite the identification of statements with weakest preconditions, we preserve the original notation, writing $wp(S, R)$ for the weakest precondition of statement S with respect to a postcondition R.

Dijkstra [Dij75] originally postulated five properties of weakest preconditions, of which all except one have subsequently been shown to be unnecessary for the refinement calculus.

Monotonicity The monotonicity property ensures that a statement which establishes a postcondition R, also establishes any weaker postcondition R'.

$$(R \Rightarrow R') \Rightarrow (wp(S, R) \Rightarrow wp(S, R'))$$

It can be shown that the set of monotonic predicate transformers form a lattice [BW89a], ordered by the refinement relation. A small language can be defined, capable of expressing any statement in this lattice; monotonicity ensures the existence of unique fixpoints of recursive equations. This shows that none of the other properties are necessary for the theory of refinement.

Feasibility An infeasible (or miraculous) statements establishes the contradictory postcondition. A statement S is infeasible when $\neg wp(S, false)$. Potentially miraculous statements have been shown to be useful in program specification [Mor87, Mor88a] and also for expressing intermediate steps in the calculation of data refinement [Mor88b].

Conjunctive A conjunctive statement establishes the conjunction of two postconditions from exactly those states in which it establishes both postconditions.

$$wp(S, R) \wedge wp(S, R') \Leftrightarrow wp(S, R \wedge R')$$

A statement which displays angelic non-determinism, establishing either R or R' as the need arises, is not conjunctive. Operationally such a statement could be implemented using back-tracking, as in Prolog (provided that the non-determinism is bounded). Such a statement is included by Back and von Wright [BW89b] and shown to be useful in data refinement [Bac89]. However, we do not consider non-conjunctive statements in this thesis.

Disjunctive Only a deterministic statement is fully disjunctive. Dijkstra showed that non-deterministic commands can be easier to reason about, for example allowing the ordering of cases in an if-statement to be ignored. This results in the property:

$$wp(S, R) \vee wp(S, R') \Rightarrow wp(S, R \vee R')$$

which holds for all monotonic statements S.

Continuity Dijkstra's continuity property:

$$wp(S, \exists n \cdot P\ n) \Leftrightarrow (\exists n \cdot wp(S, P\ n))$$

is violated by a statement with unbounded non-determinism. This arises from statements used in specification which constrain the final state to some desirable property, but have infinitely many solutions. Such statements become possible when the language is extended to allow specification.

2.2.3 Refinement of Statements

The weakest precondition defines the satisfaction relation for a statement S, where the specification is a precondition and postcondition pair.

$$S \ \mathbf{sat}\ (Pre, Post) \mathrel{\hat{=}} Pre \Rightarrow wp(S, Post)$$

A specification equivalent to $(Pre, Post)$ can be written as a statement, using a miraculous form of the generalised assigned proposed by Back [Bac88]. Using this satisfaction relation to generate a refinement relation as described above, we obtain the following standard definition for the refinement of statements:

$$S \sqsubseteq S' \mathrel{\hat{=}} \forall R \cdot wp(S, R) \Rightarrow wp(S', R)$$

2.2.4 Strongest Postcondition

Since we identify a statement with its weakest precondition, we cannot also *define* a strongest postcondition for each form of statement. Instead, the strongest postcondition

of a statement S with respect to a precondition P, written $sp(S, P)$, is defined to be the solution of the equations [Bac88]:

$$sp(S, wp(S, Q)) \Rightarrow Q$$

$$P \Rightarrow wp(S, sp(S, P)) \text{ provided } P \Rightarrow wp(S, true)$$

For a statement which may fail to terminate, these equations do not define the strongest postcondition uniquely. In this case we take the strongest solution (that is, the one with the postcondition most often false), with the result that:

$$sp(S, \neg wp(S, true)) = false$$

The strongest postcondition can be replaced by a generalised inverse of a statement [BW89b, BW93], which is itself a statement in the lattice of monotonic statements. We do not follow this approach because these inverses cannot be expressed in our language of statements.

2.2.5 First-Order Characterisation of Refinement

The *Characterisation Theorem* [Bac88] gives the following equivalence for the refinement of S by T:

$$S \sqsubseteq T \Leftrightarrow wp(S, true) \Rightarrow wp(T, sp(S, \vec{x} = \vec{x_0}))[\vec{x_0} := \vec{x}]$$

where $\vec{x}$ are the variables occurring in S or T and $\vec{x_0}$ a corresponding list of fresh variables. We only consider refinement between statements with the same sets of visible variables. Intuitively, this theorem states that S is refined by T if, in any initial state in which S terminates, executing T will reach a state which could be reached by S from this initial state.

2.2.6 Refinement in Context

We may wish to show that a statement T, which is a component of the statement S, can be replaced by T', i.e. that $S[T] \sqsubseteq S[T']$, even though $T \not\sqsubseteq T'$. The refinement of T is only correct is some *contexts*. The following two steps [Bac88] are used to show the correctness of such a refinement.

1. A context assertion Q is introduced such that:

   $$S[T] \sqsubseteq S[\{\ Q\ \}; T]$$

   To avoid proving refinements of this form repeatedly, we give a number of laws [Bac88] for context introduction. These laws allow the context to be strengthened by inference from the program structure. Context assertions can also be moved

forward through the program. In this case, a statement executed in some context gives rise to a new context when it terminates. Context assertions can also be propagated backwards, but we do not make use of this here.

For a statement S with component statement T, the context introduction is described by a law of the form:

$$\frac{\{\ Q\ \};T \equiv \{\ Q\ \};T;\{\ Q'\ \}}{\{\ P\ \};S[T] \equiv \{\ P\ \};S[\{\ Q\ \};T];\{\ P'\ \}}$$

Here, P is the initial context for S and Q is the (possibly stronger) context for the component statement T. P' is the final context for S, which may depend upon the final context Q' of T.

A context law for each statement form is given in Chapter 11 and we describe how these laws are used automatically in the refinement process. Some general laws are given below.

2. We show that T' refines T in the context Q:

$$\{\ Q\ \};T \sqsubseteq T'$$

The characterisation theorem is adapted straightforwardly for refinement in context [Bac88]:

$$\{\ Q\ \};T \sqsubseteq T' \Leftrightarrow Q \wedge wp(T, true) \Rightarrow wp(T', sp(T, \vec{x} = \vec{x_0}))[\vec{x_0} := \vec{x}]$$

Primitive Statements For any statement A, without subcomponents, the strongest postcondition can be used to propagate the context.

$$\{\ P\ \};A \equiv \{\ P\ \};A;\{\ sp(A,P)\ \}$$

In Chapter 11 the strongest postcondition of each statement is given; therefore we do not give context rules for the primitive statements.

Consequence Rule The following rule allows the context Q to be weakened, and also shows that a context introduction which applies in the context P also applies in a stronger context.

$$\frac{\begin{array}{l}\{\ P\ \};S \equiv \{\ P\ \};S[\{\ Q\ \}] \\ P' \Rightarrow P \\ Q \Rightarrow Q'\end{array}}{\{\ P'\ \};S \equiv \{\ P'\ \};S[\{\ Q'\ \}]}$$

Context Elimination For any predicate Q, Q' such that $Q \Rightarrow Q'$, we have

$$\{ Q \} \sqsubseteq \{ Q' \}$$

In particular, taking *true* for Q' this gives

$$\{ Q \} \sqsubseteq \mathbf{skip}$$

so that any context assertions can be discarded.

Chapter 3

Language Overview

An overview of the language is given in this chapter and key design choices are discussed. A more formal and complete language definition is in Part II. The language consists of the following principle syntactic categories:

Expressions and Types The types and expressions used in both specifications and programs form a single syntactic category called 'terms'; simple terms are described in Section 3.1.

Declarations New types, constants and functions can be declared, as described in Section 3.2. Section 3.3 describes datatype declarations and the special forms of term associated with datatypes.

Statements A program is made up of statements, described in Section 3.5.

Theories and Refinement Texts A theory contains a number of declarations. A 'refinement text' is formed from a specifying statement and a sequence of refinements. Theories and refinement texts are collectively known as 'units': a file contains units. Theories and refinement texts are described in Section 3.6.

Some forms of term which might be expected to be included in the language are provided by standard declarations: these are described in Section 3.4.

Syntax In this chapter (and in the rest of Part I), the language is presented using a concrete syntax. We have chosen a syntax using only ASCII characters, with similarity to existing programming languages such as Pascal and Ada. The Z specification language uses an extended character set. Although LaTeX provides a way to implement an extended character set, it requires the user to learn two syntaxes. Working directly with an extended character set (the WYSIWYG approach) greatly complicates the implementation of language processing tools. Programming languages, especially the Pascal family, have syntax which is clear rather than concise. In contrast, conciseness

is valued more in mathematical notations[1]. Some syntactic redundancy increases the chances that a simple error is detected syntactically, with the error located accurately[2]. However, to retain some of the advantages of mathematical notation, we allow new operators to be declared.

In Part II, the language is specified using an abstract syntax: the language consists of exactly those terms of the abstract syntax which satisfy the well-formation rules. The concrete syntax has also been chosen to have a direct translation to the abstract syntax. There is no need to consider the syntax given here to be definitive. Any concrete syntax which provides a representation for all well-formed terms can be used.

Lexical Rules The lexical rules distinguish four main classes of lexical token, as follows:

Class	Examples	Use
Brackets	`(  )`	
Alphanumeric	`WHERE  THEORY`	Reserved
Alphanumeric	`x  x_1  PAIR`	Unreserved
Non-alphanumeric	`;  {  }  ==>`	Reserved
Non-alphanumeric	`<=  ,  !!  +`	Unreserved
Number	`123`	

Both alphanumeric and non-alphanumeric tokens can be used as identifiers. Some identifiers of both classes are reserved. Alphanumeric reserved words are case sensitive; this avoids an unnecessary difference between reserved words and standard declarations.

A non-alphanumeric token is any sequence of non-alphanumeric characters, except the parentheses `(` and `)`. This flexible treatment of non-alphanumeric tokens allows the declaration of new operators, as described in Section 3.2.2 below[3]. Character and strings literals are additional lexical classes – see pages 22 and 204.

3.1 Terms and Types

The different forms of terms and types are shown below.

[1] The example of Chapter 4 taken from [Mor94, Chapter 21] shows the difference. Morgan uses implicit conversions and overloading to obtain a concise syntax.

[2] C++ is an example of a programming language in which the reported position of a syntax error is not always close to the mistake.

[3] These rules require spaces to be inserted unexpectedly in some character sequences; enlarging the 'brackets' token class would solve this problem.

Term	Example(s)
Sort of types	TYPE
Constant	Bool True nat_min List.length (+)
Literals	1 'b' "a string"
Variable	x y index
Application	s + 1
Function	(\x:Int @ x + 42)
Subtype	{j:Nat | j < length l} {even}
Function type	x:Int -> {y:Int | y > x}
Universal quantifier	(FORALL x: {even} @ odd (x + 1))
Existential quantifier	(EXISTS x: {even} @ x * 3 = 6)
Choice	(CHOOSE x:Nat @ x > 3)
Equality	1 = 2 (+) = (\x:Int; y:Int @ x + y)
Conditional	IF x /= 0 THEN y/x ELSE z END
Type annotation	(10 - x) : Nat
Case	CASE l OF null ==> True | cons h t ==> False END
Primitive recursion	PREC List[A]->Nat IS null ==> ... | ... END
Measure recursion	REC fact (i:Nat) : Nat IS ... MEAS i END

Case terms and primitive recursive terms are associated with datatypes and are described in Section 3.3.

3.1.1 Extension to λC

The language of terms is based on the language λC introduced in Section 2.1; however, we extend λC with some new terms. The arguments for the use of subtypes, especially in combination with dependent types, have already been presented in Chapter 1. The addition of predicate subtypes is the most important extension to λC.

To be useful as a specification language, logical operators must be included. One method of achieving this is the 'constructive' approach, based on the 'propositions as types' analogy (see Section 2.1.6). However, a constructive logic has a number of pragmatic disadvantages. Firstly, for the software engineer not necessarily at ease with classical logic, a constructive logic presents a further degree of difficultly. Secondly, proof by contradiction which is both intuitive and powerful is no longer available. Thirdly, we wish to separate program construction from proof, in contrast to the constructive approach which seeks to combine programming and proof. Instead, we extend λC to base the specification language on a classical logic.

Overall, the new forms of term are the subtype, universal and existential quantifiers, choice, equality, conditional, annotation, recursion and the forms associated with datatypes: primitive recursion and case. Some of these could be provided by declaring constants within the language rather than extending it. We use declarations as well as extensions, as described in Section 3.4. The factors determining the split be-

tween extension and declaration are detailed in Section 9.1.3. The PVS language has some additional features which could be provided by further extensions to λC — see Section 6.2.1.

3.1.2 The Type Sort

In the concrete syntax, the sort of types such as boolean and integer, is written `TYPE`, rather that the symbol $*$ used in Section 2.1.3. There is no representation in the concrete syntax for the sort $\Box$. There is no need to write this in the concrete syntax because, as noted in Section 2.1.4, $\Box$ appears only on the right hand side of a type assignment judgement.

3.1.3 Variables, Constants and Literals

Identifiers are used to stand for values. An identifier can be either a *constant* or a *variable*. Both constants and variables must be introduced before they can be used. If the constant is defined in another theory, the constant identifier may be prefixed with the theory identifier: e.g. `List.Length`.

A variable is an identifier which stands for an unknown value, for example the variable `x` in the abstraction `(\x:Int @ x + 1)` stands for any value which could be used as an argument. The type of a variable is always specified explicitly when the variable is introduced. In Barendregt's classification [Bar92] λC is *explicitly* typed. This contrasts with languages such as ML and Haskell which are *implicitly* typed; the type of a variable need not be given explicitly since it can be inferred using the type inference algorithm described by Milner [Mil78]. Explicit typing is used because no generalisation of Milner's type inference algorithm for subtypes is known.

Literals For convenience the concrete syntax of the language has special forms for some constants of the language.

`1`	is a constant of type `Int`
`'a'`	is a character literal
`"abc"`	is a string literal

The representation of character and string literals is described in Section B.8.

3.1.4 Functions and Application

The language is based on typed higher-order functions. Function can be written using a λ-abstraction. The type of a function is a 'product type', constructed using `->`. All

functions are total, so that a function of type `A -> B` maps any element of `A` to some element of `B`.

Function Term	Type
`(\x:Nat @ x + 1)`	`Nat -> Int`
`(\x:Nat; y:Nat @ x + y)`	`Nat -> Nat -> Int`
`(\a:Bool @ (\b:Bool @ not a || b))`	`Bool -> Bool -> Bool`

The function arrow `->` is right associative. Nested abstractions may be written as multiple bound variables separated by a semicolon; the following two terms are equivalent:

```
(\x:Nat; y:Nat @ x + y)      (\x:Nat @ (\y:Nat @ x + y))
```

A term representing the application of an argument to a function is written using juxtaposition: `(\x:Int @ x + 1) 2`. Function application is left associative.

Let terms A let term is syntactic sugar for an abstraction and application:

Let Term	Equivalent Term
`LET x:Int == 2 IN x + 1 END`	`(\x:Int @ x + 1) 2`

3.1.5 Subtypes and Dependent Type

Subtypes It is sometimes necessary to use a predicate to specify precisely the domain type of a function. For example, the term `{x:Int | x /= 0}` represents the type of non-zero integers. The term enclosed in braces `{ ... }` must be a predicate, that is a function to the type `Bool`. For example:

`{(<) 0}`	is the type of positive integers
`{(\x:Int @ x < 5)}`	is the type of integers less than 5
`{x:Int | x < 5}`	is an abbreviation for `{(\x:Int @ x < 5)}`

Note that subtypes and predicates have distinct syntax: if `p` is a predicate, `{ p }` is a subtype[4].

Dependent Types In the standard function type `A -> B` described above, there is no connection between the two terms `A` and `B`. However, we allow the term `B` to *depend* on `A`, so that the result type of a function depends on the value of its argument. The dependency is represented using a bound variable: `a:A -> B`. The variable `a`, which is of type `A`, may be free in the term `B`. Subtypes and dependent types can be combined: if the function `inc` has type `x:Int -> {y:Int | x < y}` then the property that any integer `z` is less than `inc z` can be inferred from the type.

[4]In contrast, PVS overloads the syntax of predicates and subtypes.

Empty Types It is possible to define a type which is empty. For example, there are no elements of type `{n:Nat | n < 0}`. A product type is empty if it has an empty range type, such as:

```
i:Nat -> j: {j:Nat | i < j} -> {n:Nat | j < n && n < i}
```

However, a product type with an empty domain is not empty, even if the range is empty.

Parametric Polymorphism The type of a function which may have arguments of any type can be represented using dependent types. In this *parametric* form of polymorphism, the type to be used is given as the first argument and the later arguments are of this type. For example, functional composition has the type:

```
A:TYPE -> B:TYPE -> C:TYPE -> (B->C) -> (A->B) -> A -> C
```

Type Correctness Conditions In the application of the term `a` to the function `f`, the term `a` must be in the domain type of `f`. If this is not the case, the application is not well-typed. Often this can be determined syntactically by the type checker. However, if the domain type of is `{x:T | p x}` and the type checker determines that the type of `a` is `T`, then it is also necessary to show that `p a` holds. The type checker produces *type correctness conditions* which are theorems which must be proved to show that the term is correctly typed. As well as for application, type correctness conditions may also be required to show that annotation, choice, conditional and equality terms are well typed.

3.1.6 Conditional Terms and Type Context

Conditional terms `IF b THEN p ELSE q END` are provided; the two alternatives `p` and `q` must be terms with the same type.

Conditional terms are one form of term where the type correctness requirements are contextual. A type correctness 'context hypothesis' may be derived either from the logical structure of the term or from type information.

Logical Context The type correctness rule for a conditional term assumes `b` when typing checking the first alternative term and its negation for the second alternative. Intuitively, this is satisfactory since the value of the conditional term only depends on the value of p when b is true. The complete set of logical context rules are shown below.

Term	Context for p	Context for q
`IF b THEN p ELSE q END`	`b`	`not b`
`p => q`		`p`
`p && q`		`p`
`p || q`		`not p`

The logical operators are defined using the conditional term – see Section 3.4.1.

Type Context The type correctness rules include context deduced from the type of one of the terms. Consider the application of a constant `x: {p}` to a function of type `{p} -> T`: the type correctness condition must show that `x` satisfies `p`, but this can be deduced from the type of x.

Asymmetry of Type Checking The use of context makes the type checking of the logical operators asymmetric, even though the operators are symmetric (i.e. commutative). For example, consider the following conjunction:

```
y > 0 && x div y < 10
```

The sub-term `x div y` gives rise to a type correctness condition to show that `y /= 0`; this follows from the context provided by the sub-term `y > 0`. However, in the term:

```
x div y < 10 && y > 0
```

the context of the sub-term `x div y` is weaker, so that it may not be possible to show that `y` is non-zero.

To see that this does not imply that conjunction is asymmetric (i.e. that $a \wedge b$ is not equivalent to $b \wedge a$, when both are well-formed), consider that the following natural deduction rules for introducing conjunction:

$$\frac{A \quad B}{A \wedge B} \qquad \frac{A \quad \begin{array}{c}[A]\\ B\end{array}}{A \wedge B} \qquad \frac{\begin{array}{c}[B]\\ A\end{array} \quad B}{A \wedge B}$$

The symmetric property of conjunction follows directly from the first rule. The second and third rules, which can be derived from the first, allow one of A or B to be assumed in the derivation of the other. The type checking for our language effectively uses the second of these rules, since the left-hand conjunct is assumed when checking the right-hand conjunct. Therefore, for the type checker to generate a provable type correctness condition it may be necessary to write the conjuncts in the correct order. But if an interactive proof tool was used for type inference, all the rules could be available, showing that the symmetry of the standard logical operators is not changed by the choice of a particular inference rule in the type checker.

3.1.7 Type Annotation

The use of subtype allows many different possible types for a term. For example, the term 2 could have any of the types:

```
Int        {x:Int | x >= 0}        {even}
```

A type annotation allows the type of a term to be specified explicitly. For example, any use of 2 can be replaced by `2 : {even}`. Since the language is explicitly typed (see Section 3.1.3), it is not often necessary to use a type annotation.

3.1.8 Equality

Two terms of the same type can be compared for equality. In `t = u`, if the type inferred for `u` is not the same as that for `t`, then `u` is implicitly annotated with the type of `t`. The comparison of a variable of a subtype `x:Nat` with a variable `y:Int` not in the subtype can be written `x:Int = y` or `y = x`.

3.1.9 Quantifiers and Choice

Logical quantifiers are provided. For example, the following predicate characterises surjective functions:

```
(\f:A->B @ (FORALL b:B @ (EXISTS a:A @ f a = b)))
```

The types used in a quantification may be empty. A universal quantification over an empty type is always true; an existential quantification is false.

The Choice Term The choice term selects an element of a type satisfying a specified predicate, if such a term exists. A quantifier syntax is used. The type of the term is the type from which the choice is taken. This type must not be empty; a type correctness condition may be generated.

Choice Term	Description
`(CHOOSE x:Nat @ even x}`	a natural, which is even
`(CHOOSE x:Nat @ x < 0}`	any natural since the property is not satisfiable
`(CHOOSE x: {n:Nat | false} @ p x)`	a type error since the type is empty

3.1.10 Measure Recursion

A recursive term introduces a bound variable which stands for a function which is defined recursively; the function is the value of the term. To ensure that there is such a function, a measure function is given. The factorial function can be represented by:

```
REC fact (i:Nat) : Nat IS
  IF i = 0 THEN 1 ELSE i * fact (i-1) END
MEAS i END
```

In this example, the identifier `fact` following the keyword `REC` is a bound variable of the term, standing for the recursively defined function. The identifier `i` stands for the arguments of the function and may be free in the body of the term. The range type of the function is given following the colon, so that the type of the term is `Nat -> Nat`.

The term following the `MEAS` keyword must be of type `Nat`. Since the bound variables standing for the function arguments may be free in the measure, the measure function `(\i:Nat @ i)` is defined. A type correctness condition is generated to show that the measure of the recursive call is less than the measure of the initial arguments. Since the measure term is in the type `Nat` it cannot be less than zero.

The measure recursion term is adapted from PVS. The requirement to use an integer measure is overly restrictive: any type with a well-founded ordering could be used, as in PVS. However, the practical necessity for this elaboration is not clear.

3.2 Declarations

3.2.1 Simple Declaration

Constants A declaration can introduce a constant in two ways: either by giving a name to a term with a type but without a definition or by giving a name to a defining term. Examples are:

```
Bool : TYPE ;  Set (A:TYPE) == A -> Bool ;
```

A constant declaration without a definition asserts that the type to which the constant belongs is not empty. Since some types are empty, a declaration may introduce an inconsistency. This is unremarkable if we consider a declaration without a definition to be a form of axiom.

Variables There is no declaration for introducing a variable. Variables only exist as bound variable in λ-abstractions and related terms. All terms which appear in declarations are *closed*, having no free variables.

Axioms and Theorems A declaration can introduce a formula which is either assumed to be true (an axiom) or can, it is conjectured, be proved to be true (a theorem)[5]. Examples appear in Section 3.4.

[5]The type checker checks axioms and theorems but does not distinguish them or use them.

3.2.2 Operator Declarations

An identifier intended to be used for a function may be declared as an operator[6]. The forms of operator are:

`INFIX p`	non-associative binary operator of precedence p
`INFIXL p`	binary operator associating to the left, with precedence p
`INFIXR p`	binary operator associating to the right, with precedence p
`PREFIX`	prefix unary operator
`POSTFIX`	postfix unary operator
`BINDER`	binder, similar to a quantifer
`LEFT`	left hand delimiter of an enumeration
`RIGHT`	right hand delimiter of an enumeration

Precedence Precedence is indicated by a single digit, with precedence 9 binding more closely than precedence 0. Unary operators always bind for closely that binary operators. If a token `Op` has been declared as an operator then `(Op)` behaves as an ordinary identifier.

Binders A binder operator is used for a function expecting another function as its argument. The syntax of terms allows a binder operator to be used in the same way as the logical quantifiers. For example, if `B` is a binder operator, then `(B x:T @ t)` stands for `(B) (\x:T @ t)`. Standard binder operators include `PAIR` and `pair`.

Delimiters These operators can be used to approximate the traditional forms of list and set enumerations. If `L` and `R` are left and right delimiters respectively, then the term `L a ; b ; c R` stands for `(L) a ((L) b ((L) c (R)))`. For example, a list can be written as `<|1; 2; 3|>`. Note that the syntax does not ensure that matching delimiters are used[7].

3.2.3 Implicit Arguments

The use of dependent types to construct polymorphic functions can result in unnecessary repetition of the type parameters. For example, functional override is represented by:

```
override (A:TYPE) (B:TYPE) (f:A->B) (a:A) (b:B) ==
  (\x:A @ IF x = a THEN b ELSE f x END)
```

[6]At present, an operator declaration must be at the start of a theory and is separate from the constant declaration: see Section 3.6.

[7]This could be done if the operator declaration associated the left and right delimiters.

If map is a function of type T -> U, t and u are terms of type T and U respectively, then override T U map t u represents the map overriden by u at the domain value t. As this example shows, it is often unnecessary to make the type arguments explicit, since they are implicit in the types of the following arguments. To declare that one or more arguments of a function are to be left implicit, the identifier is followed by the number of implicit arguments:

```
override [2] (A:TYPE) (B:TYPE) (f:A->B) (a:A) (b:B) == ...
```

The updated map can now be written override map t u. Only functions given a name in a declaration can have implicit arguments. Implicit arguments are inferred as part of type checking.

Making Implicit Arguments Explicit If there is not enough type information to allow the implicit arguments to be inferred, an argument can be given explicitly by enclosing it in brackets [...]. When a function has more than one implicit argument, explicit arguments are given left-most first. The following terms are equivalent:

```
override map t u
override [T] map t u
override [T] [U] map t u
```

Implicit Arguments and Operators Implicit arguments are particluarly useful for defining polymorphic binary operators. For example, the binary operator <+> can be defined as functional override[8]:

```
(<+>) [2] (A:TYPE) (B:TYPE) (f:A->B) (p:A#B) ==
  override f (fst p) (snd p)
```

If the implicit arguments need to be made explicit, the prefix form of the operator (<+>) must be used.

3.3 Datatype Declarations and Terms

A datatype is not a new form of term, instead a datatype declaration stands for a number of constant declarations which model the properties of the datatype. This approach is similar to that used in PVS[9] [Sha93]. There are two forms of term which can only be used with datatypes: the case term and the primitive recursion term. In

[8] Using a pair type – see Section 3.4.2.

[9] Shankar attributes this approach to datatypes to Boyer and Moore.

this respect, our approach follows Veritas$^+$ [Ver92]. In PVS, the functions created by expanding a datatype declaration include 'recursive combinator' functions which construct functions returning datatypes using primitive recursion[10].

3.3.1 Datatype Declarations

A datatype of list is declared as:

```
List [1] (A:TYPE) ::=
  null empty | cons (head:A) (tail:List [A]) nonempty ;
```

In this declaration:

- `List` is a type constructor which one type parameter. The type parameters of the datatype becomes the first argument of the functions described below.
- The functions `empty` and `nonempty` – called *recognisers* – distinguish the two list cases.
- Lists are *constructed* using the functions `cons` and `null`. The recognisers are used to characterise the type of the constructed lists.
- The functions `head` and `tail` are used to *access* the components of a list. The recognisers are used to characterise the domains of these functions.

The expansion of the list declaration is:

```
List [1] : TYPE -> TYPE ;
empty [1] : A:TYPE -> List [A] -> Bool ;
nonempty [1] : A:TYPE -> List [A] -> Bool ;
null [1] : A:TYPE -> { empty [A] } ;
cons [1] : A:TYPE -> A -> List [A] -> { nonempty [A] } ;
head [1] : A:TYPE -> { nonempty [A] } -> A ;
tail [1] : A:TYPE -> { nonempty [A] } -> List [A] ;
```

The type of a datatype component – which is the range type of the corresponding accessor function – may either be the datatype with its parameters, or must not include the datatype. A datatype case having a component with the datatype as its type is *recursive*. To ensure that a datatype is not empty, at least one of the cases must not be recursive. Here, `null` satisfies this properties, since it has no components.

[10]There is also a case term in PVS. A common method of specifying a recursive function on a datatype in PVS is to use a case term within a measure recursion, with the measure defined using the recursive combinator.

Enumerated Types Recognisers may be omitted from cases which do not have any accessor functions. An enumerated type can be declared by:

```
Colour ::= Red | Green | Blue
```

3.3.2 Primitive Recursion

The length of a list can be defined using primitive recursion:

```
length [1] (A:TYPE) == PREC List [A] -> Nat IS
                          null     ==> 0 |
                          cons h t ==> 1 + t
                       END ;
```

The PREC keyword is followed by the type of the primitive recursion term. The first part of this type must be a datatype[11], with actual parameters of the correct types. The rest of the type, here Nat is the range type of the primitive recursion. The primitive recursion term is similar to an abstraction since it always denotes a function and has a product type. The primitive recursion term has a number of cases – one for each case in the datatype. On the left hand side of the ==> symbol, a *pattern* is made up of the constructor function and a bound variable for each element of the case. The bound variables may be free in the term on the right hand side of the ==>, this term must be of the range type. The type of each bound variable is determined using the following rules:

(i) if the corresponding element of the datatype is recursive, the type of the variable is the range type of the term,

(ii) otherwise the variable type is the same as the type of the corresponding element of the datatype.

In the example, the variable h has type A, while t, which corresponds to a recursive element in the datatype, has type Int.

Dependent Types and Primitive Recursion It is possible for the function denoted by a primitive recursion to have a dependent type. For example, the function sum_max sums the elements of a list of integers, each of which does not exceed some value max. A subtype is used to specify that the sum does not exceed max multiplied by the length of the list.

[11]A subtype of the datatype cannot be used; an extension to allow this would be possible.

```
Upto (max:Nat) == {i:Nat | i <= max} ;
ListM (max:Nat) == List [Upto max] ;

sum_max (max:Nat) ==
  PREC l:ListM max -> Upto (max * length l) IS
    null     ==> 0 |
    cons h t ==> h + t
  END
```

A type correctness condition is required to show that the range type is correct.

3.3.3 Case Term

The value of a case term depends on the value of some term belonging to a datatype. For example the following term has the value `TRUE` if the term `l` is an empty list, `FALSE` otherwise.

```
CASE l OF null ==> True | cons h t ==> False END
```

The *patterns* on the left hand side of the `==>` symbol in each case introduce bound variables which may be free in the right hand side of the case. Each pattern starts with a constructors of the datatype, itself determined by the type of the term following the keyword `CASE`.

Datatype Subtypes A subtype of a datatype may be defined by excluding one or more of the constructors of the original datatype. Case term may be applied to terms which belong to such subtypes:

```
list_head [1] (A:TYPE) (l: { nonempty [A] }) ==
  CASE l OF
    cons h t ==> h
  END ;
```

A type correctness condition is required to show that the term does not belong to one of the omitted cases. Case terms are convenient but not necessary, since the recogniser and accessor functions of the datatype can be combined to have the same effect as a case term.

3.4 Standard Declarations

Some of the forms of term which might be expected to be included in the language are actually standard declarations. In this section, the following theories are described:

```
> THEORY Bool IS
>   INFIXL 3 && ;  INFIXL 2 || ;
>   INFIXR 1 => ;  INFIX  1 <=> ;

>   Bool : TYPE ; True : Bool ; False : Bool ;

>   not (a:Bool) == IF a THEN False ELSE True END ;
>   (&&) (a:Bool) (b:Bool)  == IF a THEN b ELSE False END ;
>   (||) (a:Bool) (b:Bool)  == IF a THEN True ELSE b END ;
>   (=>) (a:Bool) (b:Bool)  == IF a THEN b ELSE True END ;
>   (<=>) (a:Bool) (b:Bool) == (a => b) && (b => a) ;

>   TYPE1 == {T:TYPE | (EXISTS t:T @ True)} ;

>   AXIOM bool_disjoint == not (True = False) ;
>   AXIOM bool_complete (b:Bool) == b = True || b = False
> END
```

Figure 3.1: The Theory Bool

Theory	Description
Bool	Boolean type and operators
Pair	Pair constructors and accessors
Int	Integer type and operators

Further standard declarations are given in Appendix B.

3.4.1 Boolean Type and Operators

Figure 3.1 shows the theory Bool[12]: a boolean type is declared and the propositional operators are defined using the primitive conditional term[13]. The TYPE1 sort contains inhabited types. This is used to inform the type checker that a type is inhabited. A similar capability is primitive in PVS. A non-equality operator /= is provided as a syntactic abbreviation: it abbreviates the not function applied to an equality.

3.4.2 Pairs

In contrast to both PVS and Veritas$^+$, there is no primitive term in the language representing a pair. Figure 3.2 shows how declarations introduce both standard and

[12]Lines marked with an initial > are processed by the prototype tool to support the examples in Chapter 4.

[13]The rules for type context, described in Section 3.1.6 cannot be expressed as part of the definition and are instead built into the type checker.

```
> SEE Bool
> THEORY Pair IS
>   INFIXR 0 , # ;
>   BINDER   PAIR pair ;
>   (PAIR) [1] : A:TYPE -> f:(A -> TYPE) -> TYPE ;
>   (pair) [1] : A:TYPE -> f:(A -> TYPE) -> a:A ->
>                          (f a) -> (PAIR a:A @ f a) ;
>   fst [2] : A:TYPE -> g:(A->TYPE) -> (PAIR a:A @ g a) -> A ;
>   snd [2] : A:TYPE -> h:(A->TYPE) -> p:(PAIR a:A @ h a)
>                                   -> h (fst p) ;
>   (#) (A:TYPE) (B:TYPE) == (PAIR a:A @ B) ;
>   (,) [2] (A:TYPE) (B:TYPE) (x:A) (y:B) ==
>     (pair a:A @ B) x y ;
>   AXIOM fst_pair (A:TYPE) (f:A -> TYPE) (x:A) (y:f x) ==
>     fst ((pair a:A @ f a) x y) = x ;
>   AXIOM snd_pair (A:TYPE) (f:A -> TYPE) (x:A) (y:f x) ==
>     snd ((pair a:A @ f a) x y) : f x = y ;

>   AXIOM eq_pair (A:TYPE) (f:A -> TYPE) ==
>     (FORALL p:(PAIR) f @ (FORALL q:(PAIR) f @
>       fst p = fst q && snd p = snd q : f (fst p) => p = q))
> END
```

Figure 3.2: The Theory Pair

dependent pairs, where the type of the second element of the pair depends on the value of its first element.

The PAIR constant constructs a dependent pair type. The constructor has two arguments: the first is the type of the first element, the second is a function which gives the type of the second element depending on a value of the first element. Similarly, the constant pair is declared to construct elements of PAIR. Clearly the two elements of the pair must be arguments to the constructor; additionally the type of the second element needs to be given explicitly. Since this type may depend on the first element of the pair, the syntax must include a bound variable standing for this element.

These constructors are written using the binder syntax, with the first argument left implicit. For example, the type of a pair of Nat with the second element larger than the first and an element of this type are written:

```
(PAIR x:Nat @ {y:Nat | y > x})
(pair x:Nat @ {y:Nat | y > x}) 1 2
```

Dependent versions of the standard pair accessors fst and snd are provided. Since not all pairs require dependent types, non-dependent analogues of the type and term constructor are defined. Axioms are required to define the properties of the declarations.

```
> SEE Bool
> THEORY Int IS
>   INFIXL 6 * div mod ;  INFIXL 5 + - ;
>   INFIX  4 >= > < <= ;  INFIXL 7 ** ;

>   Int : TYPE1 ;
>   neg : Int -> Int ;
>   (+) : Int -> Int -> Int ;
>   (-) : Int -> Int -> Int ;
>   (*) : Int -> Int -> Int ;

>   (<) : Int -> Int -> Bool ;
>   (>) (i:Int) (j:Int) == j < i ;
>   (>=) (i:Int) (j:Int) == i > j || i = j ;
>   (<=) (i:Int) (j:Int) == i < j || i = j ;

>   gte_zero : Int -> Bool ;
>   Nat == { gte_zero } : TYPE1 ;
>   AXIOM gte_zero == gte_zero = (\i:Int @ i >= 0) ;
>   Pos == { x:Nat | x > 0 } : TYPE1 ;

>   (div) : Int -> { x:Int | x /= 0 } -> Int ;
>   (mod) : Int -> { x:Int | x /= 0 } -> Int ;
>   (**) : Int -> Nat -> Int
> END
```

Figure 3.3: The Theory `Int`

3.4.3 Integer Type and Operations

Integer are introduced as a primitive type. This is similar to the approach taken in the Z toolkit [Spi89] and also by Rushby *et. al.* for PVS [OSR93] where a very general number type is declared, with real, rational, integer and natural subtypes. The alternative, used in Veritas$^+$ [HDL90b], is to declare a natural number datatype and then to construct the integers (and any other numeric types) from naturals. This construction has the disadvantage that the naturals are not a subtype of the integers. Additionally, we have no mechanism for hiding the representations used for integers[14].

Naurals and Numerals Natural numbers are defined as a subtype of integers. The numeral 0, 1, 2, 3, ... are primitive terms; 0 has type `Nat`, while the other natural have type `Pos`. To avoid a circularity, the predicate `gte_zero` is declared without a definition; the required property is provided by an axiom.

[14]HOL provides such a mechanism.

Arithmetic Operators Arithmetic operations are declared. We do not give either axioms or definitions for operations. The need for axioms or definitions depends on the theorem proving technique adopted: in PVS the properties of these operators are, for the most part, specified within the decision procedures used in the prover. Division and modulus may not be applied to a zero second argument. An exponent must be a `Nat` for the result to be an integer. The use of overloading in PVS allows `+` and `*` to be redeclared for naturals, so that the sum (or product) of two naturals is a natural. This is not possible here. Unary negate is a very convenient overloading which can be implemented by the parser, representing `-1` as *neg* 1.

3.5 Statements

The different forms of statement are shown below.

Statement	Example(s)
Skip	`SKIP`
Abort	`ABORT`
Multiple assignment	`x,y := y,x`
Abstract assignment	`x,y := ANY x1,y1 WHERE x1 > y1 && y1 > x`
Precondition	`PRE x > 0 IN ... END PRE`
Sequential composition	`t := x ; x := y ; y := t`
Conditional	`IF x > 0 ==> ... OR x <= 0 ==> ... END IF`
Loop	`WHILE ... VARY N-i`
	`LOOP i < N ==> ... END LOOP`
Variable declaration	`VAR x : Nat IN ... END VAR`
Logical constant	`CON N == length l IN ... END CON`
Procedure call	`swap [x, y]`
Procedure declaration	`PROC swap IS ... IN ... END PROC`
Abstraction	`[VAL x:Nat; RES g:Nat] ...`
Application	`... [a - 1, b]`
Recursion	`REC Fact IS ... VARY x END REC`

3.5.1 Assignment Statements

Two forms of multiple assignment are provided: the standard form pairs 'left values' with terms of the same type, while an abstract assignment allows the new values of variables to specified by a predicate.

Left Values An assignment can be used to update part of a variable specified by the expression on the left hand side of the assignment symbol. The syntactic category

of the left hand side of an assignment is called a 'left value'. All left values include a variable. A left value may be:

(i) a simple variable, such as `X`,

(ii) an application to a variable of a function type, such as an array element `A i`,

(iii) an index to a variable of map type, such as a collection `C^i`, and

(iv) a component of a variable belonging to a datatype, such as the head of a list, `head L`.

Aliasing The left-values in a multiple assignment must not overlap (or *alias*), since this could result in (at least part of) the same variable being assigned two different values. The aliasing rules are summarised by the following examples:

Left values	Aliasing
`X, X`	always aliases
`X, Y`	never aliasies
`A i, A j`	no aliasing, provided `i /= j`
`A i1 i2, A j1 j2`	no aliasing, provided `i1 /= j1` or `j1 /= j2`
`head L, tail L`	never aliases

When the absence of aliasing cannot be determined statically, a correctness condition is generated.

Abstract Assignment As well as the standard multiple assignment statement, two variants of abstract assignment are included, as proposed in [BW89b]. The simpler form:

```
x,y := WHERE P x y
```

assigns new values to the variables `x` and `y` which satisfy the predicate `P`. Occurrences of `x` and `y` on the right hand side of the assignment refer to the final value of the variable. In the more general form,

```
x,y := ANY x1, y1 WHERE .....
```

occurrences of `x` or `y`[15] on the right hand side of the assignment refer to the initial value of the variable, while new identifiers `x1` and `y1` are used to refer to the final values of the assigned variables. The identifiers `x1 y1` are *bound variables* of the assignment.

[15]Note that in this form, any left-value can be used on the left hand side of the assignment. When the keyword `ANY` is omitted, only simple variables are allowed as left values.

An alternative generalisation of assignment is the specification statement introduced in [MR87, Mor88a]. The specification statement $r : [0 \leq r \mid \text{is_sqrt}\ r\ r_0 \wedge 0 \leq r]$ is equivalent to:

```
PRE 0 <= r IN
  r := ANY r1 WHERE is_sqrt r1 r && 0 <= r1
END PRE
```

Back's abstract assignment statement is preferred to Morgan's specification statement because:

(i) it is more clearly related to the familiar assignment statement,

(ii) context propagation rules rather than explicit preconditions are used to determine the context of each abstract assignment,

(iii) it is consistent with the rest of the language to make bound variables explicit, avoiding the 'zero subscript' convention, and

(iv) the assignment form with explicit bound variables allows the use of left values with abstract assignment.

3.5.2 Precondition and Context

The precondition statement has a similar form to that used in the Abstract Machine Notation [Abr91a]. The more classical form is an assertion $\{ P \}$, showing a condition which must hold at a particular point in a program. The form used here is equivalent to an assertion and a sequential composition.

Automatic Context Introduction The context of a statement is any assertion which holds immediately before the statement. At any point in a program, the context is determined automatically, using the method described in Section 2.2.6. Context assertions are introduced by preconditions, the guards in conditional and loop statements (see below) and by loop invariants. Context assertions are moved forward though the structure of the program; in particular, the context following a statement is determined using the strongest postcondition (see Section 2.2.4):

$$\{P\}\texttt{S ; T} \qquad \text{becomes} \qquad \{P\}\texttt{S ; }\{sp(S,P)\}\texttt{T}$$

Statements that can occur in more than one context do not have context automatically introduced and therefore often require an explicit precondition. The statement specifying a procedure or a recursion is treated like this.

Use of Context Automatically introduced context assertions are used to show type correctness and to show the correctness of refinements (see Section 3.6.2 below). Terms are only required to be well-typed in the context in which they are used. For example, in the guarded statement:

```
nonempty m ==> n,m := n+1, tail m
```

the term `tail m` is well-typed in the context created by the guard.

3.5.3 Declarations

Variable Declaration A new program variable can be declared using a variable declaration statement:

```
VAR x:Int; y:Int := 0 IN ... END VAR
```

A variable may be initialised: in the example `x` has no initialisation and `y` is initialsed to `0`. A program variable cannot belong to a type which is empty: if no initial value is given a proof obligation may be generated to show that the type is not empty.

To prevent types used in a program from varying, variables[16] may not be used in terms which are used as types. For example, the following statement is not allowed:

```
VAR
  x: Int ;
  y: ARRAY x Int
IN
  ....
```

Logical Constants A logical constant gives a name to a value which can be referred to but not updated. For example, the effect of a dynamically-sized array can be achieved using a logical constant.

```
VAR x: Int IN
  ....
  CON X == x IN
    VAR y: ARRAY X Int IN
      ...
```

[16]The use of formal parameters is similarly restricted, excepted as noted on page 41.

3.5.4 Procedures and Parameters

Procedure Declarations A procedure gives a name to a statement; within the scope of the procedure the name can be used as an abbreviation for the statement.

```
PROC Max IS [VAL a:Int; b:Int; RES c:Int]
  c := max a b
IN ... END PROC
```

Parameters The statement used above to define the Max procedure has formal parameters. Statements with parameters may only be used to define procedures and recursive statements. The formal parameters are variables with a type and a 'mode':

Mode	Description
VAL	Behaves as if the value of the actual parameter is 'copied into' the parameterised statement, before that statement is executed.
RES	Behaves as if the final value of the parameter is 'copied out' of the parameterised statement.
VALRES	Combines VAL and RES
REF	The formal parameter and actual parameter behave as the same variable.

The type of a RES parameter must be shown to be non-empty, like a VAR declaration. An initial value can be given or a type inhabitation condition may be required.

Procedure Call A call to the procedure Max is written `Max[x + y, y - x, x]`. The meaning of a procedure call is obtained by copy-in and copy-out substitution the actual parameters for the formal parameters in the statement which specifies the procedure[17]. The call to the Max procedure is equivalent to:

```
a, b := x + y, y - x ;
c := max a b ;
x := c
```

The actual parameters must be in the type given in the formal parameter declaration. When the formal parameter has VAL mode, the actual parameter may be any term of the correct type; for RES or VALRES modes, the actual parameter must be a left value. Further, the actual parameters for RES or VALRES modes must not alias: the rules are the same as those for multiple assignment statements.

[17]This is a simplification: a full explanation is given in Section 11.7.2.

Aliasing of REF Parameters Because copy-in and copy-out semantics are used, VAL, RES and VALRES parameters do not alias with global variables[18]. However, there is no copying of REF parameters, so a way to prevent aliasing with global variables is required. This is achieved by requiring that a variable can only be updated as an actual parameter of REF mode if the variable is *declared after* the procedure[19]. At a practical level, a programmer wishing to update an object from within a procedure body without copying has two options:

(i) declare the object first and refer to it from the procedure body as a global variable, or

(ii) declare the procedure first and refer to the object using a reference parameter.

Morgan [Mor88b] proposes eliminating reference parameters. However, all popular imperative programming languages include reference parameters, since it is not efficient to repeatedly copy large data structures[20]. In some circumstances it is possible to reference large structures as global variables and avoid passing them as parameters, however this limits the modularisation of a program.

Dependent Types in Abstractions The type of a variable in an abstraction can depend on the value of a preceding variable. This dependent typing can be used to represent Pascal's conformant arrays:

```
PROC sort IS
  [VAL len:INT16; REF list: ARRAY len INT16]
    list := ANY sorted WHERE is_perm list sorted & is_order sorted
IN ...
```

Parameters used in this way may not be updated.

3.5.5 Control Statements

The standard conditional statement of Dijkstra's guarded command language is used. The loop statement is based on the standard **do** ... **od** loop introduced by Dijkstra, but with an invariant and variant added.

[18]Examples of aliasing are given on page 174.
[19]The basis of this rule and an alternative appear in Section 11.5.5.
[20]An industrial use of the B-Tool/AMN method [HDNS96] found this problem.

```
IF  guard ==>  statement        WHILE invariant predicate
OR  guard ==>  statement        VARY variant expression
OR  ...                         LOOP  guard ==>  statement
END IF                          OR    guard ==>  statement
                                OR    ...
                                END LOOP
```

The loop invariant is a term of boolean type and the variant is a term of integer type. The result combines into a single statement all the components required to show that the loop is used correctly. A similar statement is used in the Abstract Machine Notation [Abr91a].

Recursion A recursion statement introduces a *recursive statement variable*, a statement and a variant expression which must decrease before each call to the recursive variable. For example:

```
REC Fact IS                     REF 1 BY
  PRE 0 <= n IN                   IF n = 0  ==> f := 1
    <<1>> f, n := n! , 0          OR 0 <= n ==> n := n-1; Fact;
  END PRE                                       f := f * (n+1)
VARY n END REC                    END END
```

The statement in the body of the recursion is used to specify the meaning of the recursive statement variable (here Fact) and therefore cannot mention the variable. However, the specifying statement may be refined using calls to the recursive statement variable, as shown above. A recursion can be parameterised.

3.6 Units

Units are the top-level structure in the language. A complete specification and refinement consists of a sequence of units. A unit is either a theory or a refinement text.

3.6.1 Theories

Declarations are grouped into theories which are named. Theories may not be nested. A theory may import other theories. The form of a theory declaration is:

```
SEE theoryA theoryB
THEORY new_theory IS
   operator declarations ;
```

```
    constant declarations
END
```

The theories following the keyword SEE are *directly* imported. All the constants from a directly imported theory are directly visible, that is, they are referenced as if they had been declared in the importing theory. The constants from a theory which is not directly imported[21] must be prefixed with the theory identifier: `theory_id.constant`. Further details of theories are on page 115.

3.6.2 Refinement Text

A refinement text contains a specification and a list of refinements. It is not necessary to include refinements[22]. The general form of a refinement text is:

```
SEE theoryA theoryB
SPEC specification_name IS
    statement with <<label1>> ... <<label2>>
END

REF label1 BY
    statement with <<label11>> ... <<label12>>
END
```

Refinement Labels A refinement label marks the start of a statement which has a refinement. Refinement labels are processed separately from other identfiers: any identifier or number can be used as a label. A label binds more tightly than a sequential composition. For example, in:

```
<<L1>> x := 1 ; y := 2

<<L2>> BEGIN x := 1 ; y := 2 END
```

only the assignment to `x` is labelled by `L1`, while both assignments are labelled by `L2`. Labels can be nested – a statement which is labelled may contain further labelled statements (see page 196). Statements with formal parameters may not be labelled (see page 184). An alternative sytax for refinement is proposed in Section A.6.

[21]There is no analogue to SEE to specify the set of indirectly imported theories explicitly. Theories are considered to be arranged in order and any preceeding theory, not listed as seen, is indirectly imported. The method of specifying the order is regarded as a tool issue rather than a language issue.

[22]Except when using a recursion.

Context Propagation in a Refinement Text The context is calculated automatically at each refinement, so that the refinement need only be correct in the context in which it occurs. For example, consider the following specification of a program for finding the length of a list:

```
SEE Bool Int List
SPEC ListLen IS
  VAR n:Int; l:List[Int] IN
    <<1>> n := length l
  END VAR
END
```

```
REF 1 BY
  VAR m:List[Int] IN
    n,m := 0,l ;
    <<2>> n,m := length l, nil
  END VAR
END
```

The first refinement appears to make no progress, since the statement labelled `<<2>>` is the same as that labelled `<<1>>` (except for the parallel assignment to `m`). However, the refinement of `<<2>>` is in the context created by the assignment `n,m := 0,l`, which is `n = 0 && m = l`. This context allows `<<2>>` to be refined by the loop:

```
REF 2 BY
  WHILE n = length l - length m
  VARY length m
  LOOP
    nonempty m ==> n,m := n+1, tail m
  END LOOP
END
```

In particular, the context is need to show that the loop invariant holds initially. If context was not calculated automatically, the invariant would need to be introduced explicitly, writing:

```
<<2>> PRE n = length l - length m IN
  n,m := length l, nil
END PRE
```

Complexity of Context Assertions The use of refinement is important in limiting the complexity of context assertions. In the following schematic refinement, the context for *statement2* is determined by the postcondition of *statement1*:

```
<<1>> simple statement1 ;
statement2
```

```
REF 1 BY
  complex statement3
END
```

A possible alternative would be for the context for *statement2* to be determined from the postcondition of *statement3*, which refines *statement1*. Since the statements introduced by a refinement are usually more complex that their specifications, this method of determining context would usually lead to more complex context assertions[23]. For this reason it would be less practical to use automatic context propagation to check type correctness in a language without refinement.

[23]Another issue is that, although the postcondition of *statement3* might give a stronger context, it would be unsatisfactory for the correctness of refinements to *statement2* to depend on the details of the implementation of *statement1*.

Chapter 4

Case Studies

In this chapter a number of case studies of the use of the language and tool are given. The studies are based on published examples, but have been translated into our language and checked using the prototype tool.

4.1 Largest Rectangle under a Histogram

This examples is adapted from Morgan's textbook [Mor94, Chapter 21]: the largest rectangle under a histogram. Some of Morgan's notation is overloaded and cannot be used directly in our language: equivalent definitions are formalised. The refinement steps follow the original closely; the resulting verification conditions are examined.

4.1.1 Problem Specification

4.1.1.1 Minimum of a Set

The `MinMax` theory defines a function returning the minimum of a set of natural numbers.

```
> SEE Bool Int Set
> THEORY MinMax IS
>   min_nat (s: {s:Set Nat | not (emptySet s)}) ==
>     (CHOOSE i:Nat @ s i && (FORALL j: {s} @ i <= j)) ;

>   max (i:Int) (j:Int) == IF i > j THEN i ELSE j END
> END
```

4.1.1.2 Below the Elements in a List Segment

Morgan [Mor94, Chapter 21] uses predicates of the form:

$$hs[i] \leq hs[j \rightarrow h]$$

where hs is a sequence of non-negative integers of length N, and i, j, h are integers. The interpretation of the predicate is that the ith element of hs is less than any element of the subsequence from j to $h-1$. In particular, the predicate is true if:

(i) the subsequence is empty, i.e. if $j \geq h$

(ii) i is -1 or N.

This interpretation is achieved by implicit conversions from sequences to sets, the promotion of relations to sets, so that for example $1 \leq \{2, 3, 4\}$ and by the assumption that $hs[-1]$ and $hs[N]$ are -1. The implicit conversions and overloading of $\leq$ are not possible in the language defined here[1]. Instead, the following definitions are used:

```
> SEE Bool Int List
> THEORY Nat_list_compare IS
>   INFIX 8 !? ;

>   (!?) (l:List[Nat]) (i:Int) ==
>     IF i < 0 || length l <= i THEN neg 1 ELSE l !! i END ;

>   lte_elems (l:List [Nat]) (s:Int) (f:Int) (least:Int) ==
>     (FORALL e: {elems l s f} @ least <= e)
> END
```

The function `!?` is like `!!`, returning an element of a list of naturals indexed by position when the index is valid, and returning `-1` when the index is invalid. The predicate `lte_elems l left right` is true for any value which is less than or equal to any element in the list segment from the `left` index up to the `right` index. The function `elems` (see Section B.7) is the set of elements between specified indexes of a list; the first index is zero. An index may be beyond the end of the list, possibly resulting in an empty set. In particular, the predicate `lte_elems l left right` holds if `left` and `right` define an empty list segment.

Using these definitions, we write

`lte_elems hs j h (hs!?i)` instead of $hs[i] \leq hs[j \rightarrow h]$

Note that the i is not necessarily between j and h, so that $hs[i]$ is not necessarily an element of the subsequence $hs[j \rightarrow h]$.

[1] PVS proves both overloading and a form of conversion: it would be interesting to see how much of Morgan's notation could be handled.

4.1.1.3 Largest Rectangle

The histogram is represented as a list of non-negative integers, the height of each column in the histogram. The number of elements in the histogram is made a parameter of the type.

```
> SEE Bool Int List MinMax Nat_list_compare
> THEORY Histogram IS
>   Histogram (n:Pos) == {l:List [Nat] | length l = n} ;
```

The area of the rectangle under a part of the histogram is specified by selecting the minimum element of the histogram in the given range. Note that if the range is empty or specified outside the histogram then the area is zero.

```
>   area_under [1] (n:Pos) (h:Histogram n) (l:Int) (r:Int) ==
>     IF 0 <= l && l < r && r <= n THEN
>       (r - l) * min_nat (elems h l r)
>     ELSE 0 END ;
```

Rather than defining a function similar to Morgan's *lr*, giving the area of the largest rectangle, a predicate is defined: `is_largest h l r a` means that `a` is the area of the largest rectangle which fits under the histogram `h` in the range from `l`, up to but not including `r`.

```
>   is_largest [1] (n:Pos) (h:Histogram n) (l:Int) (r:Int) (a:Int) ==
>     (EXISTS l1:Nat; r1:Nat @
>       l <= l1 && r1 <= r && (area_under h l1 r1 = a) &&
>       (FORALL l2:Nat; r2:Nat @
>         l <= l2 && r2 <= r =>
>           area_under h l2 r2 <= a)) ;
```

The largest area of an empty segment of the histogram is zero, i.e. when `r <= l` then `is_largest h l r 0`.

4.1.1.4 Properties

The properties of the definitions can be stated as theorems. For example, the theorem `lr` states the essential decomposition property of the largest rectangle which is used in the refinement: the area of the largest rectangle under a segment of the histogram is the maximum of:

(i) the length of the segment multiplied by the least element within the segment

(ii) the area of the largest rectangle in the segment before the least element

(iii) the area of the largest rectangle in the segment after the least element.

```
>   THEOREM lr (n:Pos) ==
>     (FORALL h:Histogram n; l:Int; r:Int; a:Int @
>       0 <= l && l <= r && r <= n =>
>         (is_largest h l r a <=>
>           (FORALL i:Nat; b:Nat; c:Nat @
>             l <= i && i < r && lte_elems h l r (h!!i) &&
>             is_largest h l i b && is_largest h (i+1) r c =>
>               a = max (max c b) (h!!i * (r - l))))) ;
```

The number of columns N in the histogram is introduced.

```
>   N : Pos
> END
```

4.1.2 Abstract Program

An abstract program for finding the largest area can now be given.

```
>  SEE Bool Int List MinMax Nat_list_compare Histogram
>  SPEC LargestR IS
>    VAR area : Int; hs : Histogram N IN
> <<1>> area := WHERE is_largest hs 0 N area
>    END VAR
>  END
```

4.1.3 Refinement

The refinement here follows that of [Mor94, Chapter 21]. The first step is to introduce a variable r, with the intention that r will be used to mark the end of the prefix of the histogram for which the largest area has been found: to solve the problem r will be increased to N.

```
>   REF 1 BY
>     VAR r : Int IN
> <<2>> area, r := WHERE r = N && is_largest hs 0 r area
>     END VAR
>   END
```

The next refinement step is to introduce a recursion, with three parameters:

Var	Use
i	marks the prefix of the histogram for which the largest rectangle has been found so far
b	the area of the largest rectangle from i up to j
j	marks a longer prefix of the histogram

Given that i has not reached the end of the histogram, the specification of the recursion is to:

(i) set j to an index of the histogram beyond i such that:

  (a) all the heights from i+1 up to j are greater than (or equal to) the height at i, and

  (b) the height at j is less than (or equal to) the height at i,

(ii) set b to the area of the largest rectangle in the interval from i+1 up to j.

Setting j allows the prefix of the histogram for which the largest area is known to be extended since the height at j is a new minimum. The recursion is shown to terminate by the variant N - i.

```
>   REF 2 BY
>     REC Hist IS [VAL i:Int; RES b:Int; j:Int]
>       PRE neg 1 <= i && i < N IN
>   <<3>> b, j := WHERE i < j && j <= N &&
>                       lte_elems hs (i+1) j (hs !? i) &&
>                       hs !? j <= hs !? i &&
>                       is_largest hs (i+1) j b
>       END PRE
>     VARY N - i
>     END REC [neg 1, area, r]
>   END
```

Applying the actual parameters [-1, area, r] to the recursion achieves the refinement of <<2>>. This sets r to N since this is the only value satisfying -1 < r, r <= N and hs !? r <= hs !? -1. The next refinement step introduces a loop to increase j from an initial value of i+1. The assignment within the loop is the same as <<3>>, except that the requirement to decrease the loop variant is made explicit.

```
>   REF 3 BY
>     b,j := 0, i+1 ;
```

```
>     WHILE
>       i < j && j <= N &&
>       lte_elems hs (i+1) j (hs !? i) &&
>       lte_elems hs (i+1) j (hs !? j) &&
>       is_largest hs (i+1) j b
>     VARY
>       N - j
>     LOOP
>       j < N && (i < 0 || hs !! j > hs !! i) ==>
>   <<4>> b, j := ANY b, j1 WHERE
>                    i < j1 && j1 <= N &&
>                    lte_elems hs (i+1) j1 (hs !? i) &&
>                    hs !? j1 <= hs !? i &&
>                    is_largest hs (i+1) j1 b &&
>                    N - j1 < N - j
>     END LOOP
>   END
```

Note that in [Mor94, Chapter 21] the loop guard is the simpler `hs !! j > hs !! i` but this depends on the 'out of range' elements of `hs` being defined to be `-1`. The next refinement step decomposes the problem, using the idea of theorem `lr`.

```
>   REF 4 BY
>     VAR c:Int; k:Int IN
> <<5>> c, k := WHERE j < k && k <= N &&
>                    lte_elems hs (j+1) k (hs !? j) &&
>                    hs !? k <= hs !? j &&
>                    is_largest hs (j+1) k c ;
>       b, j := max b (max c ((k - i - 1) * hs!!j)), k
>     END VAR
>   END
```

The final refinement step is to introduce the recursive call. Note that this step depends on the context `j < N` to establish the precondition of the `Hist`. The loop guard provides this context.

```
>   REF 5 BY
>     Hist [j, c, k]
>   END
```

4.1.4 Verification

The number of proof obligations generated by this example is:

```
          OBLIGATION LargestR_cc3 ==
            (FORALL hs:Histogram N; i:Int @
0             neg 1 <= i && i < N =>
1.1           i < i + 1 && i + 1 <= N &&
1.2           lte_elems hs (i + 1) (i + 1) (hs !? i) &&
1.3           lte_elems hs (i + 1) (i + 1) (hs !? (i + 1)) &&
1.4           is_largest hs (i + 1) (i + 1) 0 &&
2.1           (FORALL j:Int; b:Int @
2.2             i < j && j <= N && lte_elems hs (i + 1) j (hs !? i) &&
2.3             lte_elems hs (i + 1) j (hs !? j) &&
2.4             is_largest hs (i + 1) j b &&
2.5             j < N &&
2.6             (i < 0 || hs !! j > hs !! i) =>
2.7             (FORALL b1:Int; j1:Int @
2.8               i < j1 && j1 <= N && lte_elems hs (i + 1) j1 (hs !? i) &&
2.9               hs !? j1 <= hs !? i &&
2.10              is_largest hs (i + 1) j1 b1 &&
2.11              N - j1 < N - j =>
2.12              lte_elems hs (i + 1) j1 (hs !? j1))) &&
3.1           (FORALL j:Int; b:Int @
3.2             i < j && j <= N && lte_elems hs (i + 1) j (hs !? i) &&
3.3             lte_elems hs (i + 1) j (hs !? j) &&
3.4             is_largest hs (i + 1) j b &&
3.5             j < N &&
3.6             (i < 0 || hs !! j > hs !! i) =>
3.7             0 < N - j) &&
4.1           (FORALL j:Int; b:Int @
4.2             i < j && j <= N && lte_elems hs (i + 1) j (hs !? i) &&
4.3             lte_elems hs (i + 1) j (hs !? j) &&
4.4             is_largest hs (i + 1) j b &&
4.5             (not (j < N) || not (i < 0 || hs !! j > hs !! i)) =>
4.6             hs !? j <= hs !? i))
```

Figure 4.1: Proof Obligation for Refinement 3

Unit	Obligations
MinMax	0
Nat_list_compare	1
Histogram	3
LargestR	8

For the LargestR refinement, there are 3 type correctness conditions, all for the partial operator !!, 1 proof obligation for inhabitation of the type Histogram N and 4 for the correctness of refinements. The first refinement produces no proof obligation – it is eliminated by the simplifier described in Chapter 5 – and the second is trivial. We consider the complexity and correctness of the verification condition for one of the remaining refinements.

Refinement 3: Loop Introduction The proof obligation for the refinement of the statement labelled `<<3>>` is shown in Figure 4.1; the lines have been labelled for convenience. Following the initial line (labelled *0*) which is the context, the verification condition is divided into four sections:

1. This section shows that the loop invariant holds initially. The segment of the histogram is empty so that any element is the least element and the largest area is zero.

2. This section shows that the invariant is maintained. Most conjuncts of the invariant have been removed by the simplifier, leaving *2.12*. This conjunct follows from *2.8* and *2.9*.

3. This section shows that the variant is greater than zero when guard is true, which is stated at *3.7*; it follows from *3.5*.

4. This section shows that assigning `b` and `j` such that the invariant holds and the guard is negated is sufficient to refine the statement `<<3>>`; all the conjuncts are deleted by the simplifier except *4.6*. The loop guard *4.5* has two cases:

   (a) `j = N` which reduces *4.6* to `-1 <= hs !? i`, true for any `i`, and

   (b) `i >= 0 && hs !! j <= hs !! i`, from which *4.6* follows directly.

This analysis shows that the proof obligation is straightforward, as would be expected from the small size of the refinement steps used. The textual size of the proof obligation is increased by the hypotheses – all the lines except *1.1, 1.2, 1.3, 1.4, 2.12, 3.7* and *4.6* – this obscures but does not otherwise make the proof harder.

4.2 Integers of Finite Range

To account for the limitations of programming languages, a way to define operations on integers of finite range is required. There are a number of possibilities:

(i) use two's complement arithmetic, so that the operations are defined for all arguments but correspond to the integer operations only when the arguments are in a particular range, or

(ii) use subtypes, so that the operations always correspond to the integer operations, but can only be applied to some arguments, or

(iii) use runtime checking to detect overflow, so that the program aborts when an overflow occurs.

```
> SEE Bool Int
> THEORY INT IS
>   INFIXL 6 DIV MOD ;

>   Bits == Pos ;
>   maxint (b:Bits) == 2**(b-1) ;
>   intrange (b:Bits) (i:Int) == neg (maxint b) <= i && i < maxint b ;
>   INT (b:Bits) : TYPE1 == { intrange b } ;
>   NZ_INT (b:Bits) == { i:INT b | i /= 0 } ;
>   INT16 == INT 16 ;
>   INT32 == INT 32 ;
>   NAT (b:Bits) : TYPE1 == {i:INT b | gte_zero i} ;
>   NAT16 == NAT 16 ;

>   (DIV) [1] (b:Bits) (l:INT b) (r:NZ_INT b) : INT b == l div r ;
>   (MOD) [1] (b:Bits) (l:INT b) (r:NZ_INT b) : INT b == l mod r
> END
```

Figure 4.2: Integer Subtype Definitions

Both the second and third approaches require the absence of overflow to be proved. In the second approach, the proof is part of typechecking whereas in the third approach the proof is part of refinement. Runtime checking is used in Pascal and Ada. The first two approaches are considered in more detail below. Common definitions of integer subtypes are shown in Figure 4.2.

4.2.1 Two's Complement Arithmetic

In this approach, the arithmetic operators implement the two's complement arithmetic as used by most microprocessors. The operators can be implemented directly, but do not behave like the normal integer operators when an overflow occurs. Two's complement arithmetic is modelled by modulo arithmetic on unsigned integers, as shown in Figure 4.3. These definitions give rise to a number of type correctness conditions. For example, the annotation `i : UINT b` in the definition of `int2twos` generates the following obligation:

```
OBLIGATION int2twos_cc1 ==
  (FORALL b:Bits; i:INT b @
    i >= 0 && intrange b i => i <= maxuint b && gte_zero i)
```

```
> SEE Bool Int INT
> THEORY Int_Two IS
>   INFIXL 6 |*| ;
>   INFIXL 5 |+| |-| ;

>   maxuint (b:Nat) == 2**b - 1 ;
>   UINT (b:Nat) == {n:Nat | n <= maxuint b} ;

>   int2twos (b:Bits) (i:INT b) ==
>     IF i >= 0 THEN i : UINT b ELSE i + maxint b END ;

>   twos2int (b:Bits) (n:UINT b) ==
>     IF n < maxint b THEN n : INT b ELSE n - maxint b END ;

>   finite_op_modulo (b:Bits) (f:Int->Int->Int)  ==
>     (\l:INT b @ (\r:INT b @
>       twos2int b
>         (f (int2twos b l) (int2twos b r) mod (2 ** b)))) ;

>   NEG_T [1] (b:Bits) (l:INT b) == twos2int b
>                                    (neg (int2twos b l)) ;

>   (|+|) [1] (b:Bits) == finite_op_modulo b (+) ;
>   (|-|) [1] (b:Bits) == finite_op_modulo b (-) ;
>   (|*|) [1] (b:Bits) == finite_op_modulo b (*)
> END
```

Figure 4.3: Two's Complement Arithmetic Operators

```
> SEE Bool Int INT
> THEORY Int_Sub IS
>   INFIXL 6 !* ;
>   INFIXL 5 !+ !- ;

>   finite_op_subtype (b:Bits) (f:Int->Int->Int)  ==
>     (\l:INT b @ (\r: {r:INT b|intrange b (f l r)} @ f l r : INT b));

>   NEG_S [1] (b:Bits) (l: {l:INT b| l /= neg (2**(b-1))}) : INT b ==
>     neg l ;

>   (!+) [1] (b:Bits) == finite_op_subtype b (+) ;
>   (!-) [1] (b:Bits) == finite_op_subtype b (-) ;
>   (!*) [1] (b:Bits) == finite_op_subtype b (*)
> END
```

Figure 4.4: Arithmetic Operators using Subtypes

4.2.2 Dependent Subtypes

In this approach, dependent subtypes are used to ensure that a well-typed application of an arithmetic operator does not overflow the word length. Binary operators `!+`, `!-` and `!*` are defined in Figure 4.4, corresponding to `+`, `-` and `*`, but with a subtype constraint to prevent overflow. Note that each subtype operator `a !op b` can be rewritten to `a op b` simply by expanding the definition of `!op`.

To avoid overflow in the term `l !+ r`, the second operand `r` must belong to the subtype `{r:INT b|intrange b (l + r)}`, ensuring that the result is in the range defined by `intrange b`. This is an example of the use of dependent types: the type of `r` in `l !+ r` depends on both the first operand `l` and on the implicit parameter `b` which is the bit length.

4.2.3 Practical Usage

Both sets of operators can be used in the same program. The subtype operators should be used when the context is strong enough to prove the type correctness conditions. For example, with `x : INT32` the term `x !+ 1` gives the type correctness conditions:

```
OBLIGATION subtype_cc1 ==
  32 > 0 && gte_zero 32 => 2 <= 32

OBLIGATION subtype_cc2 ==
  1 > 0 && gte_zero 1 => intrange 32 (x + 1) && intrange 32 1
```

The first is trivial, but the second cannot be proved without further constraints on `x`. In comparison, the term `x |+| 1` gives the following type correctness conditions, both of which are trivially true:

```
OBLIGATION twos_comp_cc1 ==
  32 > 0 && gte_zero 32 => 2 <= 32

OBLIGATION twos_comp_cc2 ==
  1 > 0 && gte_zero 1 => intrange 32 1
```

The first of these TCCs arises from the subtype constraint on the `Bits` type. The second is required to show that the numeral `1` is in the type `INT32`.

4.2.4 Integer Square Root

We use the following example to examine the feasibility of discharging the type correctness conditions which arise from the use of these definition in practice. The example is

adapted from [Mor94, Chapter 9], but we refine to 'code', using the arithmetic operators on finite integers defined above.

4.2.4.1 Problem Specification

The predicate `is_sqrt` holds if the second argument is the integer square root of the first argument. The predicate `bound_sqrt` holds between an integer and a lower and (strict) upper bound on its square root.

```
> SEE Bool Int
> THEORY Is_Sqrt IS
>   is_sqrt (s:Nat) (r:Nat) == r**2 <= s && s < (r + 1)**2 ;

>   bound_sqrt (s:Nat) (lb:Nat) (ub:Nat) == lb**2 <= s && s < ub**2 ;
```

The following theorem shows how the two functions are related.

```
>   THEOREM final_bounds ==
>     (FORALL s:Nat; r:Nat @ bound_sqrt s r (r+1) <=> is_sqrt s r) ;
```

The following theorem shows that `0` and `s div 2 + 2` serve as initial lower and upper bounds on the integer square root of `s`.

```
>   THEOREM initial_bounds ==
>     (FORALL s:Nat @ bound_sqrt s 0 (s div 2 + 2))
> END
```

4.2.4.2 Refinement

The development, which is shown in Figure 4.5, follows Morgan's, with the following differences:

1. The variables are of the `NAT16` subtype and the subtype arithmetic operators are used.
2. The values suggested by the theorem `initial_bounds` are used in the initial assignment. This choice of initial values is a consequence of the use of finite integers, since the 'obvious' upper bound `s + 1` might exceed the allowed range.
3. The initialisation of `r` and q, labelled `<<3>>` establishes the context which allows the following assignment to be refined by a loop. In comparison, Morgan [Mor94, p. 65] makes the context explicit using an abbreviation.

```
> SEE Bool Int INT Int_Sub Is_Sqrt
> SPEC Int_Sqrt IS
>   VAR s:NAT16; r:NAT16 IN
>     <<1>> r := WHERE is_sqrt s r
>   END VAR
> END

> REF 1 BY
>   VAR q : NAT16 IN
>     <<2>> r,q := WHERE r + 1 = q && bound_sqrt s r q
>   END VAR
> END

> REF 2 BY
>   <<3>> r,q := WHERE bound_sqrt s r q ;
>   <<4>> r,q := WHERE r + 1 = q && bound_sqrt s r q
> END

> REF 3 BY r,q := 0, s DIV 2 !+ 2 END

> REF 4 BY
>   WHILE bound_sqrt s r q VARY  q - r
>   LOOP
>     not (q = r !+ 1) ==>
>       <<5>> r,q := ANY nr, nq WHERE
>                       bound_sqrt s nr nq && nq - nr < q - r
>   END LOOP
> END

> REF 5 BY
>   VAR p:NAT16 IN
>     <<6>> p := WHERE r < p && p < q ;
>     <<7>> r,q := ANY nr, nq WHERE
>                     bound_sqrt s nr nq && nq - nr < q - r
>   END VAR
> END

> REF 6 BY p := (r !+ q) DIV 2 END

> REF 7 BY
>   IF  s <  p !* p ==>
>     <<8>> q := ANY nq WHERE bound_sqrt s r nq && nq < q
>   OR  s >= p !* p ==>
>     <<9>> r := ANY nr WHERE bound_sqrt s nr q && r < nr
>   END IF
> END

> REF 8 BY q := p END
> REF 9 BY r := p END
```

Figure 4.5: Square Root using Finite Arithmetic

```
Type correctness condition for application in refinement labelled 3
Term: 2
Type: {r1:INT 16 | intrange 16 (s DIV 2 + r1)}
OBLIGATION Int_Sqrt_cc4 ==
  (FORALL s:NAT16 @
    2 > 0 && gte_zero 2 => intrange 16 (s DIV 2 + 2) && intrange 16 2)

Type correctness condition for assignment in refinement labelled 3
Term: s DIV 2 !+ 2
Type: NAT16
OBLIGATION Int_Sqrt_cc5 ==
  (FORALL s:NAT16 @
    intrange 16 (s DIV 2 !+ 2) => gte_zero (s DIV 2 !+ 2))

Type correctness condition for application in refinement labelled 4
Term: 1
Type: {r1:INT 16 | intrange 16 (r + r1)}
OBLIGATION Int_Sqrt_cc7 ==
  (FORALL s:NAT16; r:NAT16; q:NAT16 @
    bound_sqrt s r q && 1 > 0 && gte_zero 1 =>
    intrange 16 (r + 1) && intrange 16 1)

Type correctness condition for equality in refinement labelled 4
Term: r !+ 1
Type: NAT16
OBLIGATION Int_Sqrt_cc8 ==
  (FORALL s:NAT16; r:NAT16; q:NAT16 @
    bound_sqrt s r q && intrange 16 (r !+ 1) => gte_zero (r !+ 1))
```

Figure 4.6: Verification Conditions for Operators on Finite Integers

4.2.5 Verification Conditions

A total of 18 verification conditions are produced, of which 8 are for the correctness of refinements (the verification condition for refinement 2 is removed by the simplifier). These verification conditions are not affected by the use of finite integers. For example, the condition for refinement 6, which halves interval between upper and lower bound is:

```
OBLIGATION Int_Sqrt_cc10 ==
  (FORALL s:NAT16; r:NAT16; q:NAT16 @
    bound_sqrt s r q && not (q = r !+ 1) =>
    r < (r !+ q) DIV 2 && (r !+ q) DIV 2 < q)
```

Of the 10 type correctness conditions, 2 involve only properties of constants, as follows:

```
Type correctness condition for application in refinement labelled 6
Term: q
Type: {r1:INT 16 | intrange 16 (r + r1)}
OBLIGATION Int_Sqrt_cc11 ==
  (FORALL s:NAT16; r:NAT16; q:NAT16 @
    bound_sqrt s r q && not (q = r !+ 1) && gte_zero q &&
    intrange 16 q =>
    intrange 16 (r + q))

Type correctness condition for assignment in refinement labelled 6
Term: (r !+ q) DIV 2
Type: NAT16
OBLIGATION Int_Sqrt_cc13 ==
  (FORALL s:NAT16; r:NAT16; q:NAT16 @
    bound_sqrt s r q && not (q = r !+ 1) &&
    intrange 16 ((r !+ q) DIV 2) =>
    gte_zero ((r !+ q) DIV 2))

Type correctness condition for application in refinement labelled 7
Term: p
Type: {r1:INT 16 | intrange 16 (p * r1)}
OBLIGATION Int_Sqrt_cc15 ==
  (FORALL s:NAT16; r:NAT16; q:NAT16; p:NAT16 @
    bound_sqrt s r q && not (q = r !+ 1) && r < p && p < q &&
    gte_zero p &&
    intrange 16 p =>
    intrange 16 (p * p))
```

Figure 4.7: Verification Conditions for Operators on Finite Integers (continued)

Property	Occurrences
`not (2 = 0)`	2
`intrange 16 2`	2

Seven of the remaining eight verification conditions are shown in Figures 4.6 and 4.7 (one duplicate arising from the duplication of the subterm `p !* p` has been removed). It should be noted that since the finite integer operators are introduced only for code, the correctness conditions are only those needed to show the absence of errors in the final program.

4.3 Graph Sink

This example is based on one of two graph algorithms developed using the refinement calculus by Carrington and Robinson [CR91]: finding the sink of a graph.

4.3.1 Problem Specification

Carrington and Robinson give a Z specification: a graph is described by a schema and various functions are introduced axiomatically. In our version a graph is represented as a pair. The first element of the pair is a finite set of vertices, the second a relation on vertices. Note that this relation is between all vertices, even though a vertex is part of the graph only if it is in the set. Thus graph g has an edge between vertices v and u if and only if `fst g v` and `fst g u` and `snd g v u`. The size of the graph is the size of the set of vertices[2].

```
> SEE Bool Pair Set Int Finite
> THEORY Graph IS
>   INFIX 4 :<< ; INFIX 5 <| ;
>
>   Vertex : TYPE1 ;
>   Graph ==
>     (PAIR v:FSet Vertex @ Vertex -> Vertex -> Bool) : TYPE1 ;
>
>   graph_size (g:Graph) == size (fst g) ;
```

Operators are introduced for the subgraph relation `:<<`, and for the subgraph induced with respect to set of vertices `<|`.

```
>   (:<<) (g:Graph) (h:Graph) ==
>     (fst g :< fst h) &&
>     (FORALL v1: {fst g} ; v2: {fst g} @ snd g v1 v2 => snd h v1 v2) ;
>
>   (<|) (s:Set Vertex) (g:Graph) : Graph ==
>     (pair v:FSet Vertex @ Vertex -> Vertex -> Bool) (s /\ fst g)
>       (\v1: Vertex ; v2: Vertex @
>          s v1 && s v2 && fst g v1 && fst g v2 && snd g v1 v2) ;
```

A vertex k is a sink of a graph g if and only if there exist edges from all other vertices to k and there are no edges from k to any other vertex. Not all graphs have a sink.

```
>   sink (g:Graph) (v:Vertex) ==
>     fst g v &&
>     (FORALL u: {fst g} @ u /= v => snd g u v && not (snd g v u)) ;
>
>   has_sink (g: Graph) == (EXISTS v:Vertex @ sink g v) ;
```

4.3.2 Algorithm Refinement

The algorithm is refined in two steps:

(i) search for a candidate sink, and

[2]The requirement for the set of vertices to be finite is omitted from the Z specification.

(ii) check whether the candidate is a sink.

Note that the size of the graph is constrained: the reason for this will be apparent in Section 4.3.3.

```
>   candidate (g:Graph) (i:Vertex) ==
>     i :: fst g && (has_sink g => sink g i) ;
>
>   ok_sink (g:Graph) (ok:Bool) (i:Vertex) ==
>     (ok <=> has_sink g) && (ok => sink g i) ;
>
>   N : Pos ;
>   GraphN == {g:Graph| size (fst g) = N} : TYPE1
> END
```

If the graph g has a sink, `result` is set true and the sink is `k`. Otherwise `result` is false.

```
> SEE Bool Int Set Finite Pair Graph
> SPEC FindSink IS
>   <<D1>> VAR g:GraphN ; k:Vertex; result:Bool IN
>     <<L1>> result, k := WHERE ok_sink g result k
>   END VAR
> END
>
> REF L1 BY
>   <<L2A>> k := WHERE candidate g k ;
>   <<L2B>> result, k := WHERE ok_sink g result k
> END
```

The refinements of these two steps to complete the development of the algorithm is shown in Figures 4.8 and 4.9. The refinement is given here in slightly fewer steps than in the original article. One type correctness condition is produced and eight refinement verification conditions. As an example, the following verification condition checks the algorithm for advancing the search for a candidate sink.

```
Refinement correctness condition in refinement labelled L5
OBLIGATION FindSink_cc5 ==
  (FORALL g:GraphN; k:Vertex; t:Graph; x:Vertex @
    t :<< g && candidate t k && not (t = g) && x :: (fst g -- fst t) =>
    (snd g k x || not (snd g k x)) &&
    (snd g k x =>
      (fst t \/ singleSet x) <| g :<< g &&
      candidate ((fst t \/ singleSet x) <| g) x &&
      graph_size ((fst t \/ singleSet x) <| g) > graph_size t) &&
    (not (snd g k x) =>
      (fst t \/ singleSet x) <| g :<< g &&
      candidate ((fst t \/ singleSet x) <| g) k &&
      graph_size ((fst t \/ singleSet x) <| g) > graph_size t))
```

```
> REF L2A BY
>   VAR t : Graph IN
>     <<L3>> t,k := WHERE t :<< g && candidate t k ;
>     WHILE t :<< g && candidate t k
>     VARY graph_size g - graph_size t
>     LOOP t /= g ==>
>       <<L4>> t,k := ANY t1, k1 WHERE t1 :<< g &&
>                   candidate t1 k1 && graph_size t1 > graph_size t
>     END LOOP
>   END VAR
> END
>
> REF L3 BY
>   t,k := WHERE k :: fst g && t = singleSet k <| g
> END
>
> REF L4 BY
>   VAR x : Vertex IN
>     x := WHERE x :: (fst g -- fst t) ;
>     <<L5>> t,k := ANY t1, k1 WHERE
>                 t1 :<< g && candidate t1 k1 &&
>                 graph_size t1 > graph_size t
>   END VAR
> END
>
> REF L5 BY
>   IF snd g k x       ==> k := x
>   OR not (snd g k x) ==> SKIP END IF ;
>   t := (fst t \/ singleSet x) <| g
> END
```

Figure 4.8: Finding a Candidate Sink

The vertex `k` is a candidate sink in the subgraph `t` and the vertex `x` is chosen to extend the search by enlarging `t`. There are two cases to consider, depending on whether there is an arc from `k` to `x`. When an arc exists, it is clear that `k` is not a candidate in the enlarged graph. Since a graph has at most one sink, none of the other vertices in `t` can be a sink, so the new candidate is `x`. The other case is more straightforward.

Given the reasoning about candidate sinks (for example, the observation that there is at most one sink), it is considered unlikely that the proof of this verification condition would be fully automatic. Essentially the same proof obligation arises however the refinement step is justified, so the key property is that the verification condition is small enough to be understood. The number of refinement steps used here is sufficient to ensure that this is achieved.

```
> REF L2B BY
>   VAR s : Graph IN
>     <<L7>> s, result := WHERE candidate g k && s :<< g &&
>                              ok_sink s result k ;
>     WHILE candidate g k && s :<< g && ok_sink s result k
>     VARY graph_size g - graph_size s
>     LOOP
>       s /= g ==>
>         <<L8>> s, result := ANY s1, r1 WHERE candidate g k && s1 :<< g &&
>                          ok_sink s1 r1 k && graph_size s1 > graph_size s
>     END LOOP
>   END VAR
> END
>
> REF L7 BY
>   s, result := singleSet k <| g, True
> END
>
> REF L8 BY
>   VAR x : Vertex IN
>     x := WHERE x :: (fst g -- fst s) ;
>     <<L9B>> s, result := ANY s1, r1 WHERE candidate g k && s1 :<< g &&
>                          ok_sink s1 r1 k && graph_size s1 > graph_size s
>   END VAR
> END
>
> REF L9B BY
>   IF snd g x k && not (snd g k x) ==> s := (fst s \/ singleSet x) <| g
>   OR not (snd g x k) || snd g k x ==> result, s := False, g
>   END IF
> END
```

Figure 4.9: Checking the Candidate Sink

4.3.3 Simulation of Data Refinement

The next stage of Carrington and Robinson's development is a data refinement. We outline an approach to data refinement which could be used within our language. Since data refinement has not yet been implemented in the prototype tool, some steps are calculated by hand to determine the proof obligations and evaluate the approach.

4.3.3.1 Data Refinement

A data refinement replaces some *abstract* variables a by *concrete* variables c, maintaining an abstraction relationship R between the new and old variables. For abstract statement S and concrete statement T, data refinement is defined [BW89b] as:

$$S \sqsubseteq_{R,a,c} T \mathrel{\hat=} \forall Q \cdot R \wedge wp(S,Q) \Rightarrow wp(T, \exists a \cdot R \wedge Q)$$

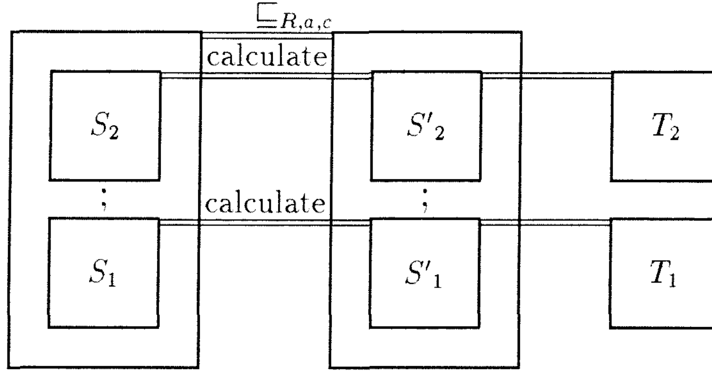


Figure 4.10: Step-by-step Data Refinement

Calculating Data Refinement A generalisation of the characterisation theorem for data refinement [BW89b] can be used to check data refinement between two statements of any form. However, it is attractive to check the data refinement in stages by calculating a data refinement [MG90, BW89b], and then to further refine the calculated statements in the standard way. Typically, at least one further refinement will be required to simplify the statements calculated under the data refinement.

For global variables v, abstract variables a and concrete variables c and refinement relation R, a data refinement calculator $\mathcal{D}_{R,a,c}$ can be defined [MG90, BW89b, Wri92] for each form of statement. For example:

$$
\begin{array}{lcl}
\mathcal{D}_{R,a,c}\,\mathbf{skip} & = & \mathbf{skip} \\
\mathcal{D}_{R,a,c}(S;T) & = & \mathcal{D}_{R,a,c}\,S; \mathcal{D}_{R,a,c}\,T \\
\mathcal{D}_{R,a,c}(b, w := b', w' \cdot post) & = & c, w := c', w' \cdot (\forall b \cdot R \Rightarrow \\
 & & \quad (\exists b' \cdot R[b, c, w := b', c', w'] \wedge post)) \\
 & & \quad \textbf{where } b \subseteq a, \textbf{ and } w \subseteq v
\end{array}
$$

The overall approach is illustrated by a refinement diagram in Figure 4.10.

Dealing with Guards Composite statements may contain elements that are not statements – notably the guards in alternation and loop statements. Since these elements may appear in the final programme, some way for the concrete versions to be specified directly is required. Moreover, calculation of concrete guards in alternation and loop statements is not straightforward[3]. Therefore, we require the user to make the concrete guards explicit when a calculated data refinement of a guarded statement is required. The following rules [But97] are used to calculate the data refinement of a

[3]Morgan and Gardiner [MG90] gives two calculated forms for guards in alternations and loops, one using an angelic command, the other with a side condition; another approach [RvW97] uses of an assumption.

loop or alternation statement and to check the guards.

$$\frac{p \wedge R \Rightarrow (b \Leftrightarrow b')}{\mathcal{D}_{R,a,c}(\{p\}\ \mathbf{do}\ b \rightarrow S\mathbf{od})\ \sqsubseteq\ \mathbf{do}\ b' \rightarrow (\mathcal{D}_{R,a,c}\ S)\ \mathbf{od}}$$

$$\frac{p \wedge R \Rightarrow (b \Leftrightarrow b')\text{ for each } i}{\mathcal{D}_{R,a,c}(\{p\}\ \mathbf{if}[\!]_i\, b_i \rightarrow S_i\ \mathbf{fi})\ \sqsubseteq\ \mathbf{if}[\!]_i\, b'_i \rightarrow (\mathcal{D}_{R,a,c}\ S)\ \mathbf{fi}}$$

Note that [Wri92, lemma 14] if I is an invariant of a loop, then $(\exists a \cdot R \wedge I)$ is an invariant of the data refinement of the loop. A concrete invariant given by the user can also be checked [Wri92, lemma 15].

4.3.3.2 Data Refinement of Graph Sink

The approach outlined above is demonstrated on the candidate sink search, using the data refinement suggested by Carrington and Robinson. The following definitions are required:

```
> SEE Bool Pair Int Function Upto Array Set Finite Graph
> THEORY VMap IS
>   GRAPH == Array N (Array N Bool) ;
>
>   vmap (g:GraphN) ==
>     (CHOOSE f:Int -> Vertex @
>       (FORALL n:Nat @ 0 <= n && n < N => fst g (f n)) &&
>       (FORALL v: {fst g} @ (EXISTS n:upto N @ v = f n))) ;
>
>   imageSet [2] (A:TYPE) (B:TYPE) (f: A -> B) (s:Set A) ==
>     (\b:B @ (EXISTS a:A @ s a && f a = b)) ;
>
>   rep_graph (g:GraphN) (A:GRAPH) ==
>     (FORALL i:upto N; j: upto N @
>       (A i j <=> snd g (vmap g i) (vmap g j)))
> END
```

The graph is represented by a boolean array. A function **vmap** mapping array indices to graph vertices is introduced, with the constraint that the map is bijective for the vertices in the graph. In the orginal article, **vmap** appears in the abstraction invariant, but is not otherwise declared.

A data refinement of the candidate sink search can now be given (refining label `D1` of `FindSink`). This is not yet supported by the prototype tool, but possible syntax is shown in Figure 4.11. For each variable declaration, concrete variables and an abstraction invariant are introduced, specifying the data refinement. Within the scope of the data abstraction, the *pattern* of the concrete statement is specified, with ... standing for the *calculated* concrete statement. The pattern should match the structure of the abstract statement.

```
DATAREF D1 VAR g, k WITH
  A : GRAPH; K : upto N
WHERE
  k = vmap (fst g) K && rep_graph g A
IS <<D2>> ... ; ... END

DATAREF D2 WITH T : Nat WHERE
  1 <= T && T <= N &&
  t = imageSet vmap (0 to T - 1) <| g
IS
   <<D2A>> ... ;
   WHILE ... VARY N - T
   LOOP T /= N ==> <<D3>> ...
   END LOOP
END

DATAREF D3 WITH X : upto N WHERE
  x = vmap (fst g) X
IS
  <<D3A>> ...
  IF  ==> <<D3B>> ...
  OR  ==> ...  END IF ;
  <<D3C>> ...
END
```

Figure 4.11: Data Refinement of the Candidate Sink Search

Calculated Data Refinement Using the data refinement calculator given above, the concrete statements can be calculated from the abstraction invariants specified in Figure 4.11. This derivation would be automated and the calculated refinement would not need to be shown to the user. For example[4], the refinement of the statement initialising t and k (see label L3 in Figure 4.8) is:

```
<<D2A>> A,K,T := ANY A1,K1,T1 WHERE
          (FORALL g:GraphN @
            rep_graph g A && 1 <= T && T <= N =>
              rep_graph g A1 && 1 <= T1 && T1 <= N &&
              singleSet (vmap g K1) <| g =
                imageSet (vmap g) (0 to T1 - 1) <| g &&
              vmap g K1 :: fst g ) ;
```

The refinements of the component statements are:

```
> REF D2A BY K, T := 0, 1 END
> REF D3A BY X := T END
> REF D3B BY K := X END
> REF D3C BY T := T + 1 END
```

[4]The complete data refinement of the candidate search has been calculated. Ordinary assignments are converted to abstract assignments, data refined and the predicate parts simplified using the rules supported by the simplifier (see Section 5.3). The resulting statement has been checked by the prototype tool.

4.3.3.3 Checking the Data Refinement

The proof obligations for the refinement of the calculated concrete statements are shown in Figure 4.12. Six trivial type correctness conditions are also produced. The proof obligations required to check the concrete guards have not been generated.

The first two proof obligations are simple and obviously valid. The obligations at labels D3B and D3C have been simplified by omitting the context. The former is trivial, but a problem arises with the obligation at label D3C. To understand the problem, consider the refinement of the statements within the candidate search loop:

```
                         Abstract                              Concrete
-----------------------------------------------------------------------------
<<D3A>> x := WHERE x :: (fst g -- fst t) ;                  X := T ;
<<D3B>> IF snd g k x          ==> k := x                    K := X ;
        OR not (snd g k x) ==> SKIP END IF ;
<<D3C>> t := (fst t \/ singleSet x) <| g                    T := T + 1
```

The context is made up of the concrete loop invariant and guard. The problem arises because the context is propagated through the *calculated* concrete statement, with the result that:

(i) the propagated context is very complex, and

(ii) the context at label D3C is not strong enough; the proof requires the context X = T, which follows from the given concrete statement X := T, but not from the calculated one.

This problem could be resolved by checking concrete statements given by the user, rather than calculating data refinements of assignment statements. The data refinement calculator would still be used to propagate the data refinement through composite statements (sequential composition, loop and if) where the abstract and concrete statements have the same structure.

```
Refinement correctness condition in refinement labelled D2A
OBLIGATION FSinkC_cc3 ==
  (FORALL A:GRAPH; T:Nat; g:GraphN @
    rep_graph g A && 1 <= T && T <= N =>
    rep_graph g A && 1 <= 1 && 1 <= N &&
    singleSet (vmap g 0) <| g = imageSet (vmap g) (0 to 1 - 1) <| g &&
    vmap g 0 :: fst g)

Refinement correctness condition in refinement labelled D3A
OBLIGATION FSinkC_cc6 ==
  (FORALL A:GRAPH; K:upto N; T:Nat @
    (EXISTS g:GraphN @
      rep_graph g A && 1 <= T && T <= N &&
      imageSet (vmap g) (0 to T - 1) <| g :<< g &&
      candidate (imageSet (vmap g) (0 to T - 1) <| g) (vmap g K)) &&
    not (T = N) =>
    (FORALL g:GraphN @
      rep_graph g A && 1 <= T && T <= N =>
      vmap g K = vmap g K && rep_graph g A && 1 <= T && T <= N &&
      imageSet (vmap g) (0 to T - 1) <| g =
      imageSet (vmap g) (0 to T - 1) <| g &&
      vmap g T :: (fst g -- fst (imageSet (vmap g) (0 to T - 1) <| g))))

Refinement correctness condition in refinement labelled D3B
OBLIGATION FSinkC_cc8 ==
  (FORALL A:GRAPH; K:upto N; T:Nat; X:upto N @
    ... && A K X =>
    (FORALL g:GraphN @
      rep_graph g A && 1 <= T && T <= N =>
      vmap g X = vmap g X && rep_graph g A && 1 <= T && T <= N &&
      imageSet (vmap g) (0 to T - 1) <| g =
      imageSet (vmap g) (0 to T - 1) <| g &&
      vmap g X = vmap g X))

Refinement correctness condition in refinement labelled D3C
OBLIGATION FSinkC_cc9 ==
  (FORALL A:GRAPH; K:upto N; T:Nat; X:upto N @
    ... =>
    (FORALL g:GraphN @
      rep_graph g A && 1 <= T && T <= N =>
      vmap g K = vmap g K && rep_graph g A && 1 <= T + 1 &&
      T + 1 <= N &&
      (fst (imageSet (vmap g) (0 to T - 1) <| g) \/
        singleSet (vmap g X)) <| g =
          imageSet (vmap g) (0 to T + 1 - 1) <| g &&
      vmap g X = vmap g X))
```

Figure 4.12: Data Refinement Proof Obligations

Chapter 5

Prototype Tool Support

A prototype tool accepting the refinement language has been implemented. The tool parses the text of theories and refinements, checks well-formation and type correctness and generates type correctness conditions, including those for type inhabitation and the absence of aliasing, and the proof obligations for the correctness of refinements[1].

The results produced by the tool are shown in the examples of Chapter 4. In this chapter the objectives achieved by developing a prototype tool are described and the design and implementation of the tool reviewed.

5.1 Objectives

The development of a tool to process the language has two objectives:

(i) validation of the language design and theory, and

(ii) validation of the approach.

Validation of the theory – that is, the well-formation and correctness rules – is achieved by testing a representation of the specification which is given in Part II. To achieve this, the tool is implemented using the Haskell functional programming language, described in the Haskell language report [PE97], so that the program resembles the specification. The representation of the specification as a Haskell program is described in more detail below. In practice, the design and specification of the language on the one hand and the development of the prototype tool on the other hand have been complementary activities. The specification provides an essential foundation for the tool, while test cases processed by the tool have shown up flaws in the specification.

[1]There is no theorem prover: a stand alone prover for the declaration and term language could be used, or the proof obligations could be converted to the language of an existing prover.

Validation of the approach requires the tool to be sufficienty usable to allow realistic examples to be tackled. Usability covers issues such as user interface, performance and error handling which are ignored in the formal specification. As a result, there is a potential conflict between these two objectives. The major usability enhancement implemented in the current version of the tool is the generation of error messages.

5.2 Design

The design is simple, consisting of the following steps:

(i) parse the source text and construct an abstract syntax tree,

(ii) check the visibility of identifiers, with the construction of a modified abstract syntax tree containing depth indices (see Section 7.3.1),

(iii) type checking, including the instantiation of implicit arguments, and

(iv) generate type correctness conditions and other proof obligations.

Examples of the code implementing each step, except parsing, is given below, noting how the implementation relates to the specification.

5.2.1 Visibility Checking

Figure 5.1 shows three cases of the visibility checking for statements, as specified in Figure 11.2. A case of the `encode_stmt` function is required for each form of the abstract syntax with identifiers (see Section 11.1). Providing there are no errors, the corresponding form of the abstract syntax with depth indices (see Section 11.2) is returned. The code in Figure 5.1 differs from its specifications in the treatment of the checks on the visibility of variables (seen, for example, in the case for `CallIS` which represents a procedure call). In the specification function $[\![S]\!]_\rho$ the checks are side-conditions, so that the specification is not defined if identifiers are not declared before use. In the code an error message is returned.

The code is structured using 'monadic programming' described by Wadler [Wad92], together with the Haskell `do` notation for monad types which is similar to list comprehension. The features of the monad used in the `encode_stmt` function are:

(i) a list of error message, and

(ii) a representation of the current position.

```
> encode_stmt g (MasgnIS ls ts) =
>   do
>     mls <- accumOk (map (encode_left g) ls)
>     mts <- accumOk (map (encode_term g) ts)
>     return
>       (do ls <- mls; ts <- mts; return (MasgnS ls ts))

> encode_stmt g (IfIS ts ss) =
>   do
>     mts <- accumOk (map (encode_term g) ts)
>     mss <- accumOk (map (encode_stmt g) ss)
>     return
>       (do ts <- mts; ss <- mss; return (IfS ts ss))

> encode_stmt g (CallIS x) =
>   case lk_id g x of
>     Procfound i -> return (Just (CallS i))
>     _           -> errMOk (proc_var_nores_error x)
```

Figure 5.1: Visibility Checking Statements

5.2.2 Type Checking

Figure 5.2 show three examples of the implementation of statement type checking, as specified in Figures 11.4, 11.6 and 11.7. The code for the `stmt_ok` function has the following difference from the specification.

1. Error messages replace side conditions. As in `stmt_encode`, a monadic structure is used for this.
2. The specification is a predicate, while the `stmt_ok` function returns[2] a statement. This is necessary since any implicit arguments of constants (see Section 3.2.3) are instantiated during type checking (see Section 9.6).
3. The 'assignable variable environment' α of Section 11.3.1 is combined into the overall environment, which is the first argument of the `stmt_ok` function.

5.2.3 Proof Obligation Generation

Proof obligations are generated to show type correctness and the correctness of refinements. The latter is determined from the weakest preconditons and strongest postcon-

[2]The full return type is `Ok (Maybe Stmt)`, where `Ok` is the monadic type constructor. The `Maybe` type constructor is used because no statement is returned when there are type errors.

```
> stmt_ok g (MasgnS ls ts) =
>   let nls = length ls; nts = length ts in
>     if nls == nts then
>       do
>         mlts <- accumOk (map (left_ok g) ls)
>         case mlts of
>           Nothing  -> return Nothing
>           Just lts ->
>             do
>               mts <- accumOk (zipWith (asgn_type_ok g) lts ts)
>               let ls = map term_to_left lts
>               a <- no_alias g ls
>               return
>                 (do ts <- mts; () <- a; return (MasgnS ls ts))
>       else
>         errMOk (masgn_len_mismatch_error nls nts)

> stmt_ok g (IfS bs ss) =
>   do
>     mbs <- accumOk (map (bool_term g) bs)
>     mss <- accumOk (map (stmt_no_fprms g) ss)
>     return
>       (do bs <- mbs; ss <- mss; return (IfS bs ss))

> stmt_ok g (CallS i) =
>   return (Just (CallS i))
```

Figure 5.2: Type Checking Statements

ditions of each form of statement. Some examples of each stage of this process are given below.

Type Correctness Conditions Figure 5.3 shows three examples of the implementation of the function *stmt_cc*, specified in Figures 11.5, 11.6 and 11.8. The examples also show the use of the implementation of the statement transformation function *asgn*, specified in Figure 11.13 and the type injection function specified in Figure 9.6. This code is also structured in a monadic style, though there are no error messages to be generated. The features of the monad used in the function `stmt_cc` are:

(i) a representation of the position in the input text, which is used as part of the final type correctness condition,

(ii) the logical context, as described in Section 11.10,

(iii) a list of correctness conditions.

```
> stmt_cc g (MasgnS ls ts) =
>   do
>     accumulate (map (left_cc g) ls)
>     accumulate (map (tcc g) ts)
>     no_alias_cc g ls
>     let u_uus = map (\(i,u) -> (u, lk_var g i)) (asgn g ls ts)
>     ptls <- accumulate (map (uncurry (inject_type g)) u_uus)
>     accumulate (zipWith
>       (\(u,uu) ptl -> poCC ptl (TccPO g Asgn u uu)) u_uus ptls)
>     return ()

> stmt_cc g (IfS bs ss) =
>   do
>     accumulate (map (tcc g) bs)
>     accumulate (zipWith
>       (\b s -> newcntxCC (andCntx b) (stmt_cc g s)) bs ss)
>     return ()

> stmt_cc g (CallS i) = return ()
```

Figure 5.3: Type Correctness of Statements

In comparison, the specification function *stmt_cc* has the logical context as a parameter and the correctness conditions[3] are as the result.

Predicate Transformers Figure 5.4 shows a fragment of the calculation of the weakest precondition of statements, as specified in Figure 11.11 and Section 11.7. This code closely resembles its specification. One difference between the specification and implementation of the weakest precondition is the use of a specialised representation for predicates – the type `Pred` – different from terms. The specialised representation of predicates allows predicates to be simplified as they are constructed; this is described in Section 5.3 below.

Refinement Figure 5.5 shows a further case of the `stmt_cc` function which is required for the refinement statement, as described in Section 12.1. The type correctness conditions for both the specifying statement and its refinement are generated, together with the correctness condition for the refinement itself. The figure also shows the use of the `impl` and `spec` functions, specified in Figure 12.2. The function `isrefinedby` (simplified in the figure) calculates the refinement correctness condition, using the 'characterisation theorem' described in Section 2.2.5.

[3] In fact only one, which is the conjunction of all the correctness conditions.

```
> wp :: Env -> Stmt -> Pred -> Pred
> wp g SkipS          q = q
> wp g AbortS         q = falseP

> wp g (PreS p s)     q = andP (termP p) (wp g s q)
> wp g (SemiS s t)    q = wp g s (wp g t q)

> wp g (MasgnS ls ts) q = substs_pred q (asgn g ls ts)
> wp g (IfS bs ss)    q = andP (foldl1 orP (map termP bs))
>   (foldl1 andP (zipWith (\b s -> impP (termP b) (wp g s q)) bs ss))
> wp g (CallS i)      q = wp g (lk_proc_var g i) q
```

Figure 5.4: Weakest Precondition

```
> stmt_cc g (RefineS l s t) =
>   do
>     stmt_cc g s
>     setposCC (LabelP l)
>       (do
>          ref <- isrefinedby g (impl s) (spec t)
>          poCC ref RefPO
>          stmt_cc g t)

> isrefinedby g s t =
>   do
>     cntx <- cntxCC
>     return
>       (impP (cntxP cntx) (impP (wp g s trueP)
>         (decby n (substs_pred (wp g' (incby n t)
>           (sp g' (incby n s) x_eq_x0)) x0_to_x))))
>  where
>        is = settolistSet (unionSet (frame g s) (frame g t))
>        n = length is
>        x0_xs :: [(Int, Int)]
>        x0_xs = zip [0..] (map (+n) is)
>        x_eq_x0 :: Pred
>        x_eq_x0 = ....
>        x0_to_x :: [Subst Term]
>        x0_to_x = [(i, VarT j) | (i,j) <- x0_xs]
>        g' = ....
```

Figure 5.5: Refinement Correctness (Simplified)

5.3 Simplication of Proof Obligations

The prototype tool does not contain a prover – apart from the difficulty of implementing a capable prover, it would not be appropriate to integrate a prover into the prototype tool, since the essence of our approach is to separate program refinement from the validation of each refinement step. However, it is appropriate to include a 'simplifier' which transforms proof obligations to a logically-equivalent simpler form. To achieve acceptable performance, the simplication rules are applied as a predicate is constructed and do not require complex pattern matching or backtracking.

The following simplications are included:

(i) discard the trivial proof obligation 'true',

(ii) evaluate the boolean operators when one of the operand is the constant 'true' or 'false',

(iii) eliminate a conjunct from the right hand side of an implication if the conjunction also appears on the left hand side,

(iv) discard quantifications over variables not free in the quantified term, providing the quantification is over a non-empty types, and

(v) eliminate variables by substitution where possible.

The following 'one-point' rules are used to eliminate quantified variables by substitution.

$$\frac{x \text{ not free in } t}{(\forall x : T \cdot x =_T t \wedge p \Rightarrow q) \equiv p[x := t] \Rightarrow q[x := t]}$$

$$\frac{x \text{ not free in } t}{(\exists x : T \cdot x =_T t \wedge p) \equiv p[x := t]}$$

We write $x =_T t$ to show that the equality must be over the type T, ensuring that t belongs to T. This is implied by $x = t$, but not necessarily by $t = x$.

5.3.1 Simplification Examples

Swapping Variables The following program refining a multiple assignment which swaps the contents of two variables illustrates the effect of simplification.

```
SEE Bool Int
SPEC SWAP IS
  VAR x : Int; y : Int IN
    <<1>> x,y := y,x
  END VAR
END
```

```
REF 1 BY
  VAR t : Int IN
    t := x ; x := y ; y := t
  END VAR
END
```

With simplication, the following proof obligation is the only one produced.

```
Refinement correctness condition in refinement labelled 1
OBLIGATION SWAP_cc1 ==
  (FORALL x:Int; y:Int @ y = y && x = x)
```

With no simplication, this proof obligation becomes:

```
Refinement correctness condition in refinement labelled 1
OBLIGATION SWAP_cc9 ==
  (FORALL x:Int; y:Int @
    True =>
    True =>
    (EXISTS x1:Int; y1:Int @ y1 = y && x1 = x && (y = y1 && x = x1)))
```

In addition, there are 16 further proof obligations for 'type inhabitation' (3), 'absence of aliasing' (5) and 'type correctness' (8). Some examples are:

```
Type inhabitation condition in specification SWAP:
  Type: Int
OBLIGATION SWAP_cc1 ==
  True => True

Type correctness condition for assignment in specification SWAP
Term: x
Type: Int
OBLIGATION SWAP_cc8 ==
  (FORALL x:Int; y:Int @ True => True)

Anti-Aliasing condition in refinement labelled 1
  Assignment to variable: x
OBLIGATION SWAP_cc14 ==
  (FORALL x:Int; y:Int; t:Int @
    (EXISTS t1:Int @ True && t = x) => True)
```

Largest Rectangle under a Histogram The number of proof obligations produced with and without simplification for the example of Section 4.1 is summarised below.

Unit	Simplication	
	Off	On
MinMax	14	0
Nat_list_compare	21	1
Histogram	120	3
LargestR	313	8

Chapter 6

Discussion

In this chapter, we survey related work and discuss the advantages and limitations of our approach compared with existing systems.

6.1 Verification and Refinement Systems

6.1.1 Program Verification Systems

Our approach has some resemblance to the 'traditional' program verification systems, based on Hoare logic. Gypsy [Amb77] and the Stanford Pascal program verifier [LS79] are two of the best known of these systems. A program is 'annotated' with assertions, including loop invariants, preconditions and postconditions for each procedure or function. A 'verification condition generator' processes the annotated program and generates logical proof obligation.

A detailed axiomatisation using Hoare-logic proof rule was carried out by London *et al.* [LGH$^+$78] for the Euclid language, an extension of Pascal with modules. The axiomatisation covers almost all the constructs of the language, which is large compared to the languages accepted by most existing refinement tools. As far as we are aware, no program verifier was developed for Euclid, but it is part of the basis for the tool-supported Ada subset SPARK [CGM92, MO94].

The characteristics of these system include:

(i) the specification language, used for annotations, is separate from the programming language,

(ii) program verification by proof of the verification conditions is separate from program development

(iii) the verification conditions tend to be large because an entire procedure body must be verified at once.

Despite considerable sophistication, verification systems based on Hoare logic have not achieved wide-spread use. One factor is the size of the verification conditions; these could be reduced if smaller development steps were verified. In addition, the specification languages of these systems have program-like type systems and are not suitable for expressing complex specifications.

6.1.2 The B-Tool: Abstract Machine Notation

Abrial's Abstract Machine Notation (AMN) [Abr89, Abr91a, Abr91b, Abr96] and its supporting B-Tool is a more advanced approach. In the B-Tool/AMN approach, a program is constructed from Abstract Machines. Each machine encapsulates a state, with an invariant and a number of operations. A machine has a specification and a number of refinements, each of which gives a new version of the operations. A single language is used for both specification and refinement, although there are some restrictions on the constructs which can be used at each stage. Only complete operations can be refined, rather than individual component statements, although the refinement steps can be made smaller by using a sequence of machines.

The B-Tool/AMN supports data refinement of machines. A novel feature of the approach is the various different ways of combining Abstract Machines, which support different degrees of information hiding.

B-Tool/AMN is based on the theory of generalised substitution rather than weakest preconditions, although the two approaches correspond closely[1]. Verification is carried out by rewriting each operation to a standard form of generalised substitution and applying a fixed refinement law to compare two standard forms. The programmer does not make use of refinement laws and the development is carried out non-interactively. Two tools supporting this approach are now available commercially[2] and commercial applications include IBM's CICS [HDNS96] and Matra Transport's METEOR project [Beh96, BDM98]. Recently a set of case studies [SS99] using the B Method has been published.

6.1.3 Compliance Analysis with DAZ

Sennett [Sen92] proposes a 'compliance notation' for demonstrating the correctness of Ada programs, written in the SPARK subset [CGM92]. The compliance notation uses the Z specification language [Spi88, Spi89] and combines features of Knuth's literate

[1]More recently [Abr96] a second set-theoretic semantics has been given, modelling predicates as sets of states and statements as functions from sets to sets.

[2]The B-toolkit developed by B-Core UK Ltd and Atelier-B developed by Steria Mediterranee.

programming [Knu84] and the refinement calculus. Subsequently, the DAZ (DERA's Ada Z) refinement tool has been developed to realise Sennett's concept.

Overview of DAZ The compliance notation is intended to be used to support a claim of compliance between a program and its specification. It aims to be readable by a third party (i.e. not the designer, but an evaluator or a maintainer) and to be flexible, so that the most critical parts of the software can be justified by the most detailed compliance argument.

In the compliance notation, Ada is extended with Morgan's specification statement and Knuth's literate programming labels. The compliance argument consists of a sequence of steps, with each step specifying a program fragment and giving a compliant implementation. Each step can be justified independently and the specifications can be formal (a specification statement) or, for less critical parts of the program, informal (the text of a label). Paragraphs of Z text, which introduce the definitions used in the specification statements, are interspersed with Ada text. The Ada declarations are translated to Z and verification conditions, also in Z, are generated to check correct refinement of specification statements by Ada statements. A detailed specification of this translation has been prepared by O'Halloran, Sennet and Smith [OSS95, OSS97].

A tool for processing DAZ has been developed by ICL Secure Systems[3] [OAK97], building on the Z theorem prover 'ProofPower'. The tool translates Ada to Z, generates verification conditions and type checks the resulting Z text. ProofPower can be used to prove the verification conditions. The DAZ tool is also used to extract the Ada program from the compliance notation; well-formation of the Ada must be checked separately (e.g. by a compiler) since the translation to Z is not intended to capture all the well-formation rules of Ada.

Several examples of the use of DAZ have been described by its developers [Smi96, OS98, OS99]. A large scale industrial evaluation has also been carried out by Praxis Critical Systems [BH97], applying DAZ to approximately 5000 lines of Ada source code, forming part of a program for which a Z specification had already been written. The compliance argument ran to approximately 750 pages.

Comparison The DAZ system has a number of similarities with the work reported here: in particular the generation of verification conditions to show the correctness of refinement steps. An important difference is the use of the existing programming and specification languages Ada and Z, allowing the DAZ tool to be applied to real development projects.

DAZ has a number of restrictions on the use of refinement compared to our approach; in particular, the left hand side of all refinement steps must be a specification statement.

[3]The DAZ and ProofPower tools are now exploited by Lemma 1 Ltd, see http://www.lemma-one.com/.

To enhance the readability of the argument, verifications conditions are restricted to a simple form, with no[4] quantifiers. This is achieved using a non-standard definition of weakest preconditions, resulting in some restrictions on the form and placement of specification statements. For example, when a specification statement is used in a sequence of statements and the postcondition refers to the initial values of updated variables, the specification statement must be the *first* statement in the sequence. It is claimed that these restriction do not limit the programs that can be shown correct – only the exact refinement steps needed to show correctness.

An interesting difference is the treatment of loops: as in our approach, the weakest precondition of a loop can be expressed without using fix-points, but in DAZ no invariant is required. Instead, a loop is introduced with a specification statement as its body; this specification statement is then refined by the implementation of the loop body. The task of choosing an appropriate specification statement is equivalent to finding an invariant[5]. Other differences include a lack of context propagation in DAZ and partial rather than total correctness[6]. Recursion is excluded from the SPARK subset of Ada, so it is not supported by DAZ.

Although the use of existing languages has many advantages, translation from one language to another (here Ada to Z) can be complex. In DAZ, the differences between the type systems in Ada and Z are overcome effectively, by mapping types that are distinct in Ada to common representations in Z. Some issues concerning scope are less easily resolved: for example, nested scopes in Ada are projected onto a flat scope in Z, so the same identifier used in two different scopes in Ada may cause a clash in the translated Z text.

6.1.4 Interactive Refinement Systems

A number of systems have been developed supporting the refinement calculus by interactive transformation using refinement rules. Typically, the language supported extends Dijkstra's guarded-command language with a form which can be used for specification: either Morgan's specification statement or Back's generalised assignment. Local blocks, types, invariants, logical constants, procedures and recursion may also be supported.

[4]Except that all the variables used in a verification condition are universally quantified at the top-level.

[5]Any statement in our language can be converted to a specification statement automatically, so it might appear that the specification statement needed as the body of a loop in DAZ could be automatically derived. However, not *any* specification statement which is refined by the Ada implementation of the loop body will do.

[6]Termination and the absence of run-time errors such as overflow can be demonstrated in DAZ by inserting appropriate assertions (see [Smi96, §4]), but it seems that the necessary assertions are not determined automatically.

Refinement Editors An early interactive refinement editor is described by Carrington and Robinson [CR90]. The tool, which is implemented using the Cornell Synthesizer Generator, is only a prototype, intended to explore the 'challenge of building support tools for the refinement calculus'. The refinement is represented in the tool as a tree, with a specification at the root and the edges corresponding to refinement steps. The basic operations of the tool are to edit and view this tree.

Refinement rules are provided in the form of transformation commands which can be applied to a selected node to extend the tree. To make a refinement, a transformation is selected and the new statement is calculated automatically. An interactive simplifier is incorporated to discharge any side conditions. Two different views of the tree are available. The first traverses the tree top-down depth-first to give a linear exposition of the refinement. The second displays only the leaf nodes and is used to extract the final program.

The authors conclude that the 'editing paradigm has been a good idea'. Their conclusions about the development of such tools include the need to able to abbreviate predicates by making definitions, the need for flexible viewing and navigation facilities similar to hypertext systems, and the importance of an integrated prover. They also believe that the user should be able to add more refinement rules and that flexible storage and retrieval of the refinement document will be required.

Red The RED refinement editor was developed by Vickers *et al.* [Vic90] at Oxford University; it is reviewed by Carrington and others [CHN+94a]. The tool supports the construction and editing of a refinement in Morgan's refinement calculus, by the application of refinement rules. The main functions supported are the display and navigation of the refinement tree, the application of refinement rules and the calculation of proof obligations arising from the side conditions of the rules. There is no theorem prover, although the proof obligations are simplified. The language can be extended by defining syntactic abbreviations and un-interpreted function symbols. Refinement sequences can be saved for reuse as rules.

The tool has a multiple window display, showing the refinement step which is the the focus of attention, the overall refinement and the proof obligations. The users enters command at a prompt and at each step a representation of the refinement tree is displayed. Although only small refinements could be tackled with this style of interface, the tool shows the different forms of display needed in an interactive tool.

RRE The RRE refinement tool [GNU92] is a more developed system, adopting a similar approach to RED. The user is presented with the program in one window and a menu of applicable refinement steps in another. A number of basic refinement rules are implemented in the primitive system; these can be combined using a tactic language to support more specialised steps. Provided the basic refinement rules are valid, a tactic with the proper form cannot give rise to an invalid refinement. The authors believe

that 'strategies for program development' can be encoded as refinement rules so that they can be reused.

The refinement system includes an integrated theorem prover which is used to simplify the program formulae as well as to discharge the side-conditions of refinement rules. The prover is based on rewrite rules, some of which are 'automatic' and others 'user-triggered'. A particular use of the latter is to expand user-defined functions, which can be introduced as declarations. The authors state that the 'rewriter needs to be closely integrated with the refinement tool'.

Data refinement of a local block is supported. The data refinement rule is parameterised by the concrete variables and the coupling invariant; from this the data refinement of the statement within the block is calculated.

Centipede The Centipede program refinement environment [BHS92] has more sophisticated viewing capabilities, based on Back's refinement diagrams [Bac91]. The refinement diagram can be expanded or collapsed to show or hide sub-derivations. This allows the user to focus on a part of the development. The program corresponding to any of the boxes in the diagram can also be viewed.

A structured editor is used to select a component statement for further refinement, which can be specified either manually — the user enters the new statement using the editor – or by selecting a transformation rule from a menu. Side conditions are checked by exporting a verification condition to the HOL theorem prover.

6.1.5 Refinement using Window Inference

Grundy [Gru92] describes a refinement tool making use of a 'window inference' extension to the HOL prover. Window inferencing allows the proof of a transitive and reflexive relation between two terms by successive transformations of subterms. To transform a term, a subterm is selected and a new window opened. Each window has a *focus*, which is a term being transformed, the relation being preserved and a set of assumptions. The rules which open a window may generate new assumptions from the *context* in which the focus occurs. Any theorem of the logic can be used to simplify the focus of a window, using the assumptions that are part of the window.

Window inference is considered to be better suited to program refinement than goal-oriented proof methods because the final result of the refinement does not need to known in advance. Window inference allows large terms – such as a program – to be generated by successive transformations of subterms. Contextual information is also handled neatly.

In Grundy's application of window inference to refinement, programs are represented as predicates, based on the approach of Hoare [Hoa84] and Hehner [Heh89, Heh90]. This

allows programs to manipulated as terms in the logic and the refinement relation to be implication.

Grundy notes that it is not necessary to adopt the 'programs are predicates' approach in order to use windows inference for refinement. This is confirmed by von Wright [Wri94], who describes a program refinement system built on the HOL theorem prover using weakest precondition semantics and window inference. A language of commands is embedded in the HOL logic by declaring HOL constants which represent the commands as predicate transformers, equal to the weakest preconditions of the commands in Back's version of the refinement calculus. This 'shallow embedding' of the refinement language in HOL means that statements *are* terms in the HOL syntax, allowing functions and types declared in existing HOL libraries to be used directly. A disadvantage is that the statement syntax is non-standard. In particular, an assignment command is represented by a state to state function, written using a lambda abstraction, and the state is a tuple.

The refinement rules are proven as theorems and window inference is used, so that the complete refinement is itself a theorem in the HOL logic. Data refinement is possible, using the method of abstraction commands [BW89b] to distribute the data refinement through the structure of the program, so that each subcomponent can be refined separately.

6.1.6 The Refinement Calculator

The Refinement Calculator [BLRvW95, BL96, BGL+97] has been developed at Åbo Akademi and Southampton University, based on von Wright's system described above.

The basis of the Refinement Calculator is program transformation using refinement rules. A user-friendly interface [LRvW95] to the HOL window inference package is used, so that the user works with the standard syntax of statements, including program variables and assignment statements. The statement to be refined can be selected using the mouse and refinement rules are selected from a menu, without the use of HOL commands in ML. Some knowledge of HOL is still required to discharge proof obligations. The refinement is still fully checked by the HOL prover. A transcript of the proof can be output in HTML in a hierarchical format that allows subderivations to be expanded or hidden.

Apart from the user interface, two notable features of this tool are:

(i) a transformation which automatically propagates context assertions through the structure of a statement, to the point where a refinement which depends upon the context is to be made, and

(ii) the use of total correctness assertions to allow flexible refinement rules, in particular for loop introduction, which allow the user to choose the form of the refined statement, rather that it being determined by pattern matching.

Published future plans [BGL+97] are for extensions to support procedures, modules and action systems.

6.1.7 The Program Refinement Tool

The Program Refinement Tool [CHN+95, CHN+96] has been developed by Carrington *et al.* at the University of Queensland, following experience with the prototype tool described above.

PRT is based on the Ergo theorem prover. Since Ergo does not support higher-order logic, predicate transformers cannot be represented in the way that they are in HOL. Instead, a modal logic is defined so that some terms have an implicit dependence on the state [CHN+94b]. This requires some concepts to be defined by axioms which are primitive to the HOL logic, so that an equivalent demonstration of soundness is not achieved. This aside, PRT provides a fully-proved refinement, similar to the Refinement Calculator: Ergo is used to prove refinement rules and to discharge any side-conditions.

The method of window inference, which is used in Ergo, has been elaborated to deal with program context by Nickson and Hayes [NH96]. Context predicates are a distinguished form of hypothesis in the inference window and are automatically propagated by the window opening rules.

Program window inference also handles invariants (for program variable types, using the method of Morgan [Mor89]) and syntactic side-conditions, such as the disjointness of assigned variables. Tactics are used to discharge syntactic side-conditions automatically.

6.2 Languages with Dependent Types

A number of languages with subtypes and dependent types have been developed, particularly as logical notations within theorem proving systems, using the 'propositions are types' and 'proofs are programs' analogies of constructive mathematics. Examples of such systems include Nuprl [CAB+86] and Coq [CH88].

Our language draws particularly on the Veritas+ and PVS systems, both of which are interactive theorem provers for classical logics. The languages supported by these systems are reviewed below. A theorem prover for dependent type theory has also been built on top of the HOL theorem prover by Jacob and Melham [JM93].

6.2.1 PVS

The language of the PVS theorem prover PVS [ORS92] is based on a classical higher-order logic with subtypes and dependent types. The support tools for the language,

described in the reference manual [OSRSC98], are integrated into the theorem prover. Apart from this, the pragmatics of the language are similar to the approach described here: for example, ASCII-only syntax is used and a type checker produces type correctness conditions.

The principal difference between the language described here and that of PVS is that in the latter types depend on terms, *but terms do not depend on types*. Terms and types are separate syntactic categories: in particular, function application applies to terms not types. Instead, parametric polymorphism is provided at the level of theories: theory parameters may be either types or terms. Actual parameters are given when a theory is imported. Multiple instances of a theory are permitted in any context, with the result that constants are overloaded: the overloading is resolved using the types of the theory parameters. Ad-hoc overloading of identifiers is also allowed, providing that the constants have different types. It is also possible to specify the theory parameters explicitly as part of a reference to a constant from an imported theory.

The PVS language includes a number of features not provided in the language described here, but which could be included if required. These include:

Tuple Types A tuple type constructor of arbitrary length is a primitive form in PVS. Function and tuple types can be combined so that the type of a function from a pair `[[t1, t2] -> u]` can also be written as `[t1, t2 -> u]` and the domain types may have bound variables which can be referenced in the range type.

Records Record types and terms are provided. Two record types are equal if they have the same field names and types, no matter the order of declaration.

Tables Functions may be specified by cases using the multi-way conditional `COND` term, which is nearly equivalent to nested `IF-THEN-ELSE`[7]. The `TABLE` term allows the specification of a function by cases in a tabular format. The `TABLE` term can be expanded using `COND` or, for datatypes, `CASES`.

Type Judgements A type judgement is a theorem about types which provides additional information to the type checker. Type correctness conditions are generated which check that the theorem is valid. There are two forms:

1. `HAS_TYPE` applies to a constant. When the constant has a function type, the judgement states a closure condition. For example, integer addition is closed on the natural numbers. For constants of other types, the `HAS_TYPE` judgement gives the constant a type other than one given by its declaration. Examples are:

```
JUDGEMENT HAS_TYPE 4 (even?)
JUDGEMENT HAS_TYPE + [nat, nat -> nat]
```

[7]The `COND` term requires the conditions to be disjoint; this is not required in a nested conditional.

2. SUBTYPE_OF states a relationship between two subtypes. The following judgement would be used to allow a function on nzint to be applied to a posint, without generating a type correctness condition:

```
posint : NONEMPTY_TYPE = {n:int | n > 0} CONTAINING 1
nzint : NONEMPTY_TYPE = {n:int | n /= 0} CONTAINING 1
JUDGEMENT posint SUBTYPE_OF nzint
```

Type Conversions These are functions which are automatically inserted by the type checker to correct a type mismatch. A simple example arises when a function F of type [[Nat -> Int] -> Int] is applied to a term g of type [Int -> Int]. Since F expects a function which can be applied to a natural number yielding an integer, one would expect g to be satisfactory. However, the type of g must be subtype of the domain type of F, implying that the domain types of g and F must be *equal*. The solution is to convert g to a function on the natural numbers in the obvious way. In PVS, the conversion is declared in a theory as follows:

```
restrict [T: TYPE, S: TYPE FROM T, R: TYPE]: THEORY
 BEGIN
  f: VAR [T -> R]
  s: VAR S
  restrict(f)(s): R = f(s)
  CONVERSION restrict
 END restrict
```

The same phenomenon applies in the language described here. The restrict function – which must be applied *explicitly* – is declared as follows:

```
restrict [3] (A:TYPE) (B:TYPE) (p:A->Bool) (f:A->B) ==
 (\a:{p} @ f a)
```

Inductive Definitions Sets can be defined inductively. For example, the declaration:

```
even(n:nat): INDUCTIVE bool =
  n = 0 OR (n > 1 AND even(n - 2))
```

defines even to be the smallest subset of the naturals satisfying

```
(FORALL (n:nat): n = 0 OR (n > 1 AND even(n - 2))
   IMPLIES even (n))
```

so that a number is even *if and only if* it has the property used in the induction definition. Occurrences of the set being defined within the defining property must be *positive*; if necessary type correctness conditions are generated to check the monotonicity of the property with respect to the subset ordering.

Semantics of PVS Owre and Shanker [OS97] have written a formal semantics for a subset of the PVS language. The semantics covers function and tuple types, predicate subtypes, dependent types, conditional expressions and parametric theories. Other language features such as records, recursive definitions, datatypes, tables, type judgements and conversions are not covered. The treatment of parameteric theories is restricted, since parameters are assumed to be constants or type declarations *without definitions* and assumptions are not covered. Moreover, all references to names from a parameteric theory must have explicit parameters, so that overloading and name resolution are not described.

The semantics is functional: a partial function τ assigns a type to a preterm with respect to a context. This function has side conditions giving the decidable checks and the proof obligations for type correctness with a clear distinction between the decidable and undecidable checks. A meaning function $\mathcal{M}$ is also defined and both type and logical soundness are proved.

6.2.2 Veritas

The Veritas$^+$ [HDL90a, HDL90b] system implements a higher order logic with dependent types and subtypes. Originally developed for hardware verification, it is based on Martin-Löf's Intuitionistic Type Theory (ITT) and conventional higher-order logic. Unlike ITT, the reasoning is classical.

The type constructors include the dependent function type (Π generalising $\rightarrow$) and the dependent cartesian product (Σ generalising the cartesian product $\times$). All functions are total. Subtypes are constructed from a type and a characteristic predicate $\{ \nu : \sigma \mid \phi \}$. Types are terms, with the type of a *bool* being U_0, the smallest in a hierarchy of type universes $U_0, U_1, \ldots$. Polymorphism is achieved by the use of types as parameters, with the type of subsequent parameters depending on earlier ones. Just as described in Section 3.2.3, the type parameters can be elided in context where their values can be inferred from the context. In Veritas, but not in the language described here, the type of a bound variable may also be elided.

Datatypes are supported as primitive type constructors, rather than using the 'shell principle', although datatypes can only be declared at the top-level and not, for example, within a λ-abstraction. Functions on datatypes are constructed using primitive recursion only; general recursion is not supported.

Implementation The implementation of Veritas$^+$ [FNG92] draws on the LCF tradition of theorem provers, and on the Curry-Howard 'Propositions as Types' principle. The terms of the Veritas language are defined by 'term formation rules'. Adapting the LCF approach for forming theorems, the formation rules are implemented by the constructor functions of an abstract datatype. The inference rules of the Veritas logic are the formation rules of a type of 'derivations'. Derivations are not theorems, but

record the proof of a theorem. As expected from the 'Propositions as Types' principle, the constructors for derivations are similar to the term constructor. However, in Veritas the two sets of constructors are distinct. Just as the 'sort' of a term is a type (which is term), the sort of a derivation is the theorem (also a term) which the derivation establishes.

The formation rules for both terms and derivations are encoded in an executable meta-language, forming an executable definition of the logic which is the basis of the Veritas tool. A particular feature of this approach [HD86] is that the meta-language is purely functional. This requires the signature of terms and derivations to be handled as first-class values, rather than to be a global data-structure which is updated by side-effects of declarations. The signature contains declarations, definitions and axioms. It seems that the languages Standard ML, MIRANDA and Haskell have been used as the meta-language in different versions of the system.

Type Checking and Logical Inference Since subtypes can be formed, type checking is not decidable in Veritas. Some term formation and inference rules taken from the Veritas documentation [Ver92] illustrate how term formation and logical inference are connected. In the following, ν is a symbol, τ is a term, σ is a type (a term of sort U_i), ϕ is a formula (a boolean term), and ζ is a derivation. The rules for forming a subtype term and injecting a term into a subtype are as follows.

$$\frac{[\![\nu : \sigma]\!]\phi}{\{\nu : \sigma | \phi\}}\ \boxed{\text{Subtype}} \qquad \frac{\tau : \sigma \quad \{\nu : \sigma|\phi\} \quad \zeta : \phi[\tau/\nu]}{\tau : \{\nu : \sigma|\phi\}}\ \boxed{\text{Injection}}$$

The subtype injection rule is used to change the type of a term to a subtype. One of the components of the formation rule is a derivation ζ which proves the theorem that the term satisfies the predicate that characterises the type. An inference rule (i.e. a derivation formation rule) allows the subtype relationship between types to be inferred. This can be used in a formation rule for application[8].

$$\frac{\zeta : (\forall \nu : \sigma \cdot \exists \nu' : \sigma' \cdot \nu = \nu')}{(\textbf{st-intro}\ \zeta) : (\sigma \subseteq \sigma')}\ \boxed{\text{Subtype-intro}}$$

$$\frac{\tau_1 : (\Pi \nu : \sigma_1 \cdot \sigma_2) \quad \tau_2 : \sigma_3 \quad \zeta : (\sigma_3 \subseteq \sigma_1)}{(\tau_1\ \tau_2) : \sigma_2[\tau_1/\nu]}\ \boxed{\text{Application}}$$

Note that this characterisation of the subtype relationship $\subseteq$ depends on the terms τ, τ' in $\tau = \tau'$ *not* being required to have the same type (or even the same base type) for the equality term to be well-formed.

The term formation and inference rules provide a very rigorous formal definition of the Veritas system in a way which does not exist for PVS. However, a denotational

[8]This is an elaboration of the rule given in the Veritas documentation [Ver92], where the types σ_1 and σ_3 are equal. However, the formation rule 'Widen' combined with the simpler rule for application gives the formation rule shown here.

semantics comparable to that of PVS, which could be used to argue the soundness of the Veritas inference system, does not seem to exist.

Using Veritas Using the formation and inference rules described above, term construction and type checking is, in principle, carried out interactively. This approach allows a user to prove subtype membership when required. It is stated that simpler cases can be automated using tactics, but the level of manual assistance required in practice is not clear. It is possible that something similar to our type checker could be constructed for the Veritas logic using a tactic for goal-directed term formation, where our 'type correctness conditions' are the undischarged hypotheses of the term formation. However, this approach might give rise to unprovable hypotheses in some circumstance where our type checker gives type error messages (see Section 9.5), which is a more useful response.

6.3 Comparison and Evaluation

In the section, we evaluate our approach in comparison with existing systems.

6.3.1 Using a Universal Refinement Law

A principle feature of our approach is the use of a single law for proving refinements. Compared to program derivation by transformation using a *refinement calculus*, the development method and intuition of our approach are closer to those used for non-formal program development. In particular, the traditional edit-compile style of working is preserved, with development separated from verification. The constraints of this approach must be considered in the language design. We consider these issues in the following sections.

6.3.1.1 Refinement Laws and Intuition

The examples of Chapter 4 demonstrate that the characterisation theorem provides a practical basis for showing the correctness of a program by refinement, allowing flexibility in the size of the refinement steps. To compare the use of refinement laws with our approach, consider the refinement derivation shown in Figure 6.1 which uses the refinement laws from Morgan's book [Mor94, Appendix C]. In this derivation, it is assumed that the precondition $x = x_0 \wedge y = y_0$ can be left implicit, even though it is needed to discharge the side condition of the final step:

$$\begin{array}{l} t = t_0 \wedge x = x_0 \wedge y = y_0 \Rightarrow \\ \qquad x + y = x_0 + y_0 \wedge y - x = y_0 - x_0 \end{array}$$

$x, y := x + y, y - x$
$\sqsubseteq$ *Abbreviation 8.1, Simple specification*
  $x, y : [x = x_0 + y_0 \wedge y = y_0 - x_0]$
$\sqsubseteq$ *Law 6.1, Introduce local variable*
  $|[$ **var** $t : \mathbb{Z} \cdot$
    $t, x, y : [x = x_0 + y_0 \wedge y = y_0 - x_0] \quad \lhd$
  $]|$
$\sqsubseteq$ *Law 3.5, Following assignment*
  $t, x, y : [t = x_0 + y_0 \wedge y = y_0 - x_0]; \quad \lhd$
  $x := t$
$\sqsubseteq$ *Law 3.5, Following assignment*
  $t, x, y : [t = x_0 + y_0 \wedge y - x = y_0 - x_0]; \quad \lhd$
  $y := y - x$
$\sqsubseteq$ *Law 5.2, Assignment*
  $t := x + y$

Figure 6.1: Derivation of Variable Swap

This corresponds to the proof obligation generated by our prototype tool when this refinement is made in a single step (see Section 5.3.1).

Although derivation at this level of detail may not always be necessary with interactive refinement tools[9], the derivation shows that the overall refinement is sometimes much more intuitive that the sequence of laws which justify it. This applies particularly to the laws which manipulate assignment statements and initial values. In our approach refinement laws are still important, but are used for educating programmers and developing their intuition for correct refinement, rather than for direct transformation of formulae.

6.3.1.2 Context Propagation

Automatic propagation of context assertions (using the rules of Section 2.2.6) is an important features of our approach. To show the effect of context propagation we

[9]In particular, the variable swapping could be provided as a derived rule or the total correctness assertions of the Refinement Calculator could be used.

```
  <<2>> r,q := WHERE q = r + 1 && bound_sqrt s r q

REF 2 BY
  <<3>> r,q := WHERE bound_sqrt s r q ;
  <<4>> r,q := WHERE q = r + 1 && bound_sqrt s r q
END

REF 4 BY
  WHILE bound_sqrt s r q VARY q - r
  LOOP
    q /= r + 1 ==>
      <<5>> r,q := ANY nr, nq WHERE bound_sqrt s nr nq &&
                                    nq - nr < q - r
  END LOOP
END
```

Figure 6.2: Example of Context Propagation

reproduce a small extract of Morgan's [Mor94] square root case study[10]:

$$\begin{array}{ll} & q, r : [r^2 \leq s < q^2 \wedge r + 1 = q] \\ \sqsubseteq & I \mathrel{\hat{=}} r^2 \leq s < q^2 \cdot \\ & q, r : [I \wedge r + 1 = q] \\ \sqsubseteq & q, r : [I]; \\ & q, r : [I,\ I \wedge r + 1 = q] \quad \lhd \\ \sqsubseteq & \text{"invariant } I\text{, variant } q - r\text{"} \\ & \mathbf{do}\ r + 1 \neq q\ \rightarrow \\ & \qquad q, r : [r + 1 \neq q,\ I,\ q - r < q_0 - r_0] \\ & \mathbf{od} \end{array}$$

In the above, the precondition to establish the loop invariant must be made explicit. This is achieved with brevity by using the macro definition I and by using a specification statement with an invariant predicate $[\ldots,\ I, \ldots]$. The final refinement step is not a simple match to the law for introducing an iteration: the expression $r+1 = q$ is matched against $\neg GG$ and the bound on the variant is omitted[11], so the formulae produced by a tool might be more complex that the textbook version.

Our version of the refinement is shown in Figure 6.2. In our notation, the refinement labelled `<<3>>` establishes the context in which `<<4>>` is refined to a loop: there is no need to make the precondition explicit. Similarly, no explicit precondition on the statement in the loop is needed to use the context established by the guard. Another difference is that we cannot define a *macro* such as $I \mathrel{\hat{=}} r^2 \leq s < q^2$ but instead use the

[10]The same example is also used in Section 4.2, but with variables of an integer subtype instead of $\mathbb{N}$.

[11]As noted by Morgan.

function `bound_sqrt`, which must have explicit arguments. The verification condition for refinement `<<4>>` is trivial[12]. As a result, `<<5>>` could be replaced by its refinement with little added complexity.

Context and Type Checking Context is fully integrated with type checking of terms. As described in Section 3.1.5, terms also give rise to context. The context arising from statements and terms are handled in a uniform way, so that terms are only required to be type correct in the context in which they are used.

Context Assumptions Back and von-Wright [BW98, §28] gives rules for propagating context assumptions[13] through the structure of programs, similar to the rules for context assertions. It it possible that context assumptions could be of use within our approach. A possible use would be to overcome a limitation of the automatic propagation of context assertions: context is not propagated into recursion blocks (or loops). The difficulty is to identify the predicates which could validly be propagated. This could be achieved using a context assumption within the recursion block, propagated backwards to its start.

Both the Refinement Calculator and PRT support context propagation. In the Refinement Calculator context is represented as assertion statements and a single command propagates assertions though any statement. In PRT, context is implicit and is propagated automatically by window opening rules. However, this is believed to be the first use of automatic context propagation in a non-interactive tool.

6.3.1.3 Complexity of Correctness Conditions

A possible concern about the use of a universal refinement law is the complexity of the verification conditions produced. The case studies of Chapter 4 demonstrate that the 'acceptable' verification conditions can be produced. This depends on having small refinement steps and simplifying verification conditions. Small refinement steps are available in our method, so that the user can trade-off simpler verification conditions against more complex refinements. This is an advantage compared to the Abstract Machine Notation, where refinement is always between machines.

Verification Method Refinement provides a means to develop verified programs, but, in our approach, the verification method is not fixed. At least three options are available:

[12]Except to show that the variant is bounded below by zero, which can be inferred from the guard and the loop invariant (provided that the variables are of type $\mathbb{N}$).

[13]Back notes that context assumptions provide an equivalent capability to the Morgan's invariants [Mor89], discussed in Section 6.3.2.2.

(i) consider the refinement to be the main correctness argument and review the refinement text directly,

(ii) review the verification conditions,

(iii) carry out proof, either automatically or interactively.

It is likely that a combination of these verification methods should be used. The relative cost-effectiveness of the different methods is not known and should be investigated empirically. In contrast, in our experience with the B-Tool/AMN (using the B-Core version)[14], the verification method was fixed, using automatic proof first, followed by interactive proof if necessary. Pretty printing of verification conditions was not provided to allow review, presumably because of their complexity.

Refinement Laws It is instructive to consider refinement laws for the introduction of a loop. Morgan's law for introducing an iteration [Mor94] (here simplified to a single alternative) is:

$$w : [inv, inv \wedge \neg b] \sqsubseteq \textbf{do } b \rightarrow w : [inv \wedge b, inv \wedge (0 \leq V < V_0)] \textbf{ od}$$

The guard and invariant are determined from the form of the specification statement being refined, whilst the variant expression must be given when the refinement law is invoked. There is no side condition. This contrasts with the laws described by Butler and *et al.* [BGL+97] for the Refinement Calculator[15]. The transformation 'Loop Introduction 1' is:

$$\frac{\begin{array}{l}\vdash pre \Rightarrow inv \\ \vdash inv \wedge \neg G \Rightarrow post[v' := v]\end{array}}{\begin{array}{l}\vdash \{pre\}; v := v' \cdot post \sqsubseteq \\ \quad \textbf{do } G \rightarrow \{inv \wedge G\}; v := v' \cdot inv[v := v'] \wedge E[v := v'] < E \textbf{ od}\end{array}} \quad \boxed{v \text{ not free in } post}$$

In contrast to Morgan's law, both the guard, the invariant and the variant must be given when the law is invoked, with the advantage that the form of the predicate in the nondeterministic assignment being refined is not restricted. The cost is that there is now a side condition. It is clear that there is a trade-off between the flexibility of the law and the need for side conditions. A second loop introduction law is also given by the same authors, in which the statement within the loop is also supplied when the law is invoked, using a total correctness assertion in the side condition. This second law closely approximates our approach.

How close to Morgan's law for introducing an iteration with no proof obligation can we come? By taking small refinement steps and choosing the form of terms appropriately, the proof obligation for a loop can *almost* be eliminated. For example, returning to the

[14]The capabilities of the latest version may have improved.

[15]Morgan's law may also be available.

integer square root example and adjusting[16] the loop introduction steps at refinement <<4>>, results in the following proof obligation for the loop introduction.

```
(FORALL s:Nat; r:Nat; q:Nat @
   bound_sqrt s r q && 0 < q - r =>
   (FORALL q1:Nat; r1:Nat @
     bound_sqrt s r1 q1 && 0 < q1 - r1 && not not (q1 = r1 + 1) =>
     q1 = r1 + 1))
```

Only the simplification rule $\neg\neg p \equiv p$ would be required to eliminate this proof obligation altogether.

6.3.1.4 Batch versus Interactive Language Processing

The use of a universal refinement law makes it possible to process the refinement text using a batch program, similar to a compiler. This is simple, familiar to programmers and allows flexibility in the use of proof or alternative verification methods. The proof tool is separate from the statement language processing tool, and needs to support only the declarations and terms of the language.

Separation of Well-formation The use of a batch tool allows the checking and reporting of well-formation errors in the refinement text to be separated from any proof functions. This contrasts both with refinement editors and with B-Tool/AMN. In the latter, the user syntax is transformed to the underlying 'unsugared' language before most errors are reported.

Interactive Tool Support Although we argue that interactive program transformation and integration with a theorem prover are not essential to the use of refinement, we do not argue that interactive tool support is without value. It is now standard for conventional languages to be supported by an Interactive Development Environment (IDE), supporting functions such as program viewing and editing, cross referencing, incremental compilation, and symbolic debugging. A refinement IDE enhanced to take account of the refinement hierarchy and including a theorem prover would be very useful.

[16]The changes are slightly more complex than expected. As described in Section 11.8.4, the form of loop proof rule used here differs from Morgan's. To eliminate the proof obligation condition $I \wedge \bigvee_i B_i \Rightarrow 0 < E$ by simplification, the predicate $0 < E$ asserting that the variant is bounded must be added to the invariant, in turn requiring the variant to be one larger than necessary. It would be easier to introduce loops without proof obligations if Morgan's proof rule were used as the basis of our loop statement.

6.3.2 Language Features – Statements

6.3.2.1 Distinguishing Features

Since our language of statements is based on a selection of the extensions to the Dijkstra's guarded command language introduced by Back, Morgan and others, it has only minor distinguishing features. The forms of statement common for the refinement calculus, including recursion and procedures are all supported.

Left Values Our language allows the use of complex 'left values' in assignment statements such as `a i, a (i+1) := a (i+1), a i`. Complex left value allow assignment to a component of variable with a function type (such as, but not only, arrays), map type (including a collection) and datatypes. Left values may also be used in procedure calls. The well-formation conditions for left values in multiple parameter lists are specified to avoid aliasing.

Types and Subtypes Program variables can be declared using any type constructor including subtypes, with the constraint that the type of a program variable must not depend on another variable.

Dependent Types Pascal's conformant array parameters can be represented using dependent types. However, unlike in Pascal, this capability arises as a consequence of the more general capabilities of dependent types. Similarly, Ada's unconstrained array and discriminated records can be modelled using dependent types.

Pass-by-Reference Parameters Reference parameters are commonly used by programmers to avoid inefficient copying of data. We have proposed a particularly simple condition on the use of reference parameters, which is sufficient to ensure that monotonicity is preserved.

6.3.2.2 Comparison with Other Refinement Languages

Guards The refinement calculus [Mor94] includes guarded command such as $b \Rightarrow S$ which can be used independently of loops and conditional statements. The AMN language includes a construct (SELECT) expressing the non-deterministic choice of guards. Guarded statements are miraculous when the guard (or, for SELECT, all the guards) are false. Such a statement could easily be added to our language, but it is not clear that it would be useful in a purely top-down refinement methodology.

Invariants Morgan [Mor89] describes 'invariants' in the refinement calculus. An invariant, a predicate on the program state which is true for all statement within a specified scope, can express a type constraint. However, since arbitrary relationships between variables can be used as invariants, the approach is more general than the type checking provided here, although the main application appears to be to check subtype properties. An advantage claimed for invariants is that they are assumed rather than asserted and are therefore automatically maintained, if necessary by miracles. This avoids the need to check that types are correct until the final stage of the development, when potentially miraculous statement must be removed. In contrast, our approach requires type checking at each stage of the development. Invariants of this form are support by the Program Refinement Tool described above.

It is likely that invariants of this form could be used within our framework, but it would not be beneficial to do so in conjunction with our type system, which provides direct support for subtypes. In contrast, Morgan uses a type system similar to that of Z [Spi89], in which subtype constraints can be expressed but are checked during proof rather than by the type checking.

6.3.2.3 Loop and Recursion

The use of the characterisation law to justify refinements has some consequences on the language design presented in this thesis. To use the characterisation theorem, each statement must have a weakest precondition and a strongest postcondition. This is straightforward, except for loops and recursion. The standard approach ([Mor94, Chapter 23], [BW98, Chapter 20]) is to define the loop and recursion in terms of the least-refined fixed points of a function from statements to statements. Since it is not convenient to reason directly using fixed points, the definitions are used to justify refinement laws.

When refinement laws are used, the invariant and other components of the loop are either inferred from the form of the statement being refined or proposed by the user. Here, we take a similar approach to the B-Tool/AMN described above, insisting that a loop statement is accompanied by an invariant predicate and a variant expression. Since the variant is of an integer type, it may not be possible to show the termination of a loop if the body of the loop is not a continuous statement, as noted by Boom [Boo82]. Statements which are not continuous exist in our language, but this problem seems unlikely to arise in practice[17]. Despite the apparent differences, the practical application of our approach is likely to be very similar to that of the refinement calculus.

The treatment of recursive statements in our language is even more similar to that of the refinement calculus. In the refinement calculus [Mor94, Chapter 13], the variant expression n and a logical constant N standing for the initial value of the variant are

[17] PVS allows ordinals to be used show termination of recursively defined functions; this approach could be applied here.

made explicit in the syntax of a recursion block:

$$\mathbf{re}\ X\ \mathbf{variant}\ N\ \mathbf{is}\ n \cdot S$$

In this context, a recursive call can be introduced as a refinement of a statement matching the specification of the recursive label:

$$\{\ N > n \geq 0\ \}S\ \sqsubseteq\ X$$

The only difference between this and our approach is that, here, the logical constant and the assertion needed to show that the variant has decreased are introduced automatically. The recursion block is exceptional among the statements of our language, since it can only be used in conjunction with a refinement. However, the loop statement could be presented as a special case of recursion. We sketch the idea, for simplicity ignoring the loop variant. The statement:

```
WHILE inv LOOP b ==> S END LOOP
```

could be defined to be equivalent to the *refinement*:

```
REC loop IS
  PRE inv IN <<1>> x := WHERE inv && !b END PRE

REF <<1>> BY IF b THEN S; loop ELSE SKIP END IF END
```

This shows that the loop with invariant (and variant) can be considered to encapsulate a refinement, just as the recursion does. At the syntactic level, some way to determine the statement being refined by the loop is required. We have used the invariant and variant which, together with the loop guards, provide the components of a non-deterministic assignment statement which is taken as the loop specification; in principle, any form of statement could be used.

6.3.2.4 Conventional Imperative Language Features

Compared to conventional imperative languages, some common language features are still missing from our language.

Control Statements A greater variety of control statements, beyond the loop and conditional, is common in imperative programming languages. Examples are case statements, for-loops and loops with exits. Our approach could be extended with these statements[18].

[18]The forms of loop with exits might have to be restricted.

Exceptions The use of exceptions in the refinement calculus is described by King and Morgan [KM95]. This suggests that exceptions could be supported in the refinement approach described here. However, it is noted that to give the semantics of statements with exceptions, King and Morgan elaborate the weakest-precondition predicate transformer, making it likely that a substantial revision to the specification of our language would be required.

Functions Conventional programming languages include program *functions*, which are similar to procedures but can be used as as terms. Functions that do not update the global state are supported in the SPARK Ada subset and the approach used there could be applied to extend our language.

Procedure Parameters Pascal and related languages[19] allow procedures to be parameterised with procedures. Support for procedure parameters in our language appears achievable. The language specification associates a procedure name with the specifying statement, rather as a variable name is associated with its type. Thus a statement could be the type of a formal parameter of procedure type and the refinement relation could be used for type containment.

Modules Abrial's B-Tool/AMN demonstrates that a verifiable language can include comprehensive 'modular' constructs.

Object-Oriented Features The mechanisms of class extension by inheritance are not *monotonic* with respect to refinement, since an extended class cannot necessarily be implemented using the implementation of its base class. For this reason, the addition of object-oriented features to the refinement calculus is still an open problem. However, it is notable that some of the B-Tool/AMN machine combinators are not monotonic, but are still useful for constructing specifications.

6.3.2.5 Program Transformation and Angelic Non-Determinism

Two limitations of the approach described here follow from the use of a single built-in rule for refinement, based on the characterisation theorem. Firstly, our approach is more suitable for top-down refinement than for more general program transformation. Secondly, the use in our language of statements with angelic non-determinism is precluded[20].

Program Transformation Some techniques of *program transformation*, which may be based on refinement, extend the top-down approach, in the sense that transformations are applied to possibly large program fragments generated by earlier transformations. Although our approach allows 'nested refinement' (i.e. refinement of a part of a

[19] It is notable that Ada83 is an exception, although procedures can parameterise 'generic' packages.
[20] More precisely, such statements could be introduced but there would be no way to eliminate them.

program introduced by an earlier refinement), it is likely to result in large verification conditions. It is possible that our approach could be extended to provide better support for transformational techniques.

Ward [War92] describes a recursion removal transformation. Using transformation, the derivation of iterative algorithms proceeds from a recursively defined specification, to a recursively defined procedure and then, by recursion removal, to an iterative algorithm. This approach is demonstrated on the Schnorr-Waite graph marking algorithm, with a data refinement applied before the recursive removal step [War96]. Gravell [Gra91] describes a technique for refining abstract program which are then specialised by introducing appropriate assumptions. He notes that this technique applied in a purely top-down manner does not lead to efficient programs and that transformations are required following specialisation. Butler [But97] describes transformations from trees enriched with paths to trees implemented by pointer structures.

Angelic Non-determinism Back and von Wright [BW89a] introduce an angelic assignment statement $\overline{v := v' \cdot p}$. This statement is the dual of the non-deterministic assignment used here, in the sense that it is strict (i.e. non-terminating) rather than miraculous and the non-determinism is angelic rather than demonic. Back [Bac89] presents a method for data refinement using statements with angelic non-determinism. Operationally, angelic non-determinism can be considered to represent back-tracking. Statements with angelic non-determinism can also be used to structure a program refinement [Hes94].

6.3.3 Subtypes and Dependent Types

A novel feature of our refinement language is its type system. In the introduction, we explained how the choice of subtypes and dependent types was motivated by the potential for efficient theorem proving in the presence of 'partial' functions. In this section, we review the use of the type system, especially for representing programming languages and consider the practical acceptability of type correctness proof obligations. The properties of our type checker are reviewed and some criticisms of predicate subtypes are examined.

6.3.3.1 Programming Language Features

Subtypes and dependent types allow programming language features that are normally represented by partial functions to be represented as total functions. Finite range numbers appear in the example of Section 4.2; arrays, pointer and strings can also be modelled (see Sections B.4, B.6 and B.8).

Type Checking in Context in Programming Language The use of context in type checking (see Sections 3.1.5 and 9.4.5) can be realised in a programming language by lazy evaluation of the propositional connectives. For example, since `q` is type checked assuming `p` in the conjunction `p && q`, `q` should only be evaluated when `p` evaluates to true. Logical operators in the C programming language have this behaviour; Ada also provides lazy (or left-right) logical operators.

λ-abstractions in Programming Languages The use of a language based on the typed λ-calculus provides a way to represent expressions which require special treatment in standard programming languages. For example, since an array is a function, a λ-abstraction provides a convenient representation for an array value:

```
(\x: upto 10 @ IF x = 0 THEN 0 ELSE
               IF x :: 1 to 4 THEN 1 ELSE 2 END END)
```

'Aggregate' expressions in Ada are similar to this, except that the bound variable is implicit.

Array Range Errors The use of subtypes matches the constraints of a programming language. Consider the example of an array access. In a programming language any occurrences of `A i` where the index `i` is out of range can cause non-termination even if the effect of the statement does not depend on the value of the array at the illegal index. It is therefore appropriate to show the absence of any such accesses by type checking.

Definedness in the Refinement Calculus In Morgan's refinement calculus [Mor94, §6.7], undefined expressions do not cause statements to abort. For example, the refinement:

$$x : [x = 1/0] \sqsubseteq x := 1/0$$

is valid: $1/0$ is a number, even if we do not know which one. This provides a convenient development method and uniform semantics (in particular, the statement on the left hand side of the refinement terminates, so the one on the right must too).

Since division operators in programming languages do not have the required property, the statement $x := A/B$ is not code. Instead, partial expression are code only when accompanied by a suitable assertion, here:

$$\{\ B \neq 0\ \}; x := A/B$$

In our approach, these same assertions would be sufficient to discharge the proof obligations for type correctness. One advantage claimed by Morgan is that assertions are only required for code rather than for every occurrence of a partial expression. As the

example of Section 4.2 shows that partial expressions can be introduced as the final refinement step, so that the number of checks required is similar. It is not known if Morgan's approach has been implemented: the description given above would need to be elaborated to allow for all the contexts in which partial functions can be used.

Arrays in HOL In an example derivation using the Refinement Calculator [BGL+97, 'Sorting an Array'], Butler *et al.* use a HOL array type $(\alpha)array$ which is polymorphic on the type α, with the following functions:

$$size : (\alpha)array \rightarrow num$$
$$lookup : (\alpha)array \rightarrow num \rightarrow (\alpha)array$$

In HOL all functions are total, so $lookup\ a\ i$ is a num whether or not $i < (size\ a)$. As a result, it may be possible to prove that a program which accesses an array out of range is correct. Taking Morgan's view, it could be said that $lookup\ a\ i$ is only code in the context of $i < (size\ a)$. In the derivation, $lookup\ a\ i$ is only used in this context, but the HOL type system alone does not ensure this.

6.3.3.2 Type Correctness Conditions and the Use of Subtypes

The examples of Chapter 4 show that subtypes can be used without type correctness conditions arising in unacceptable quantity or of unacceptable complexity. Reducing the number of type correctness conditions requires care by the user. Some guidelines are:

(i) the arguments of a function should belong to a subtype only when the type correctness of the function's definition requires this and not just because the intended use of the function suggests the use of a subtype,

(ii) subtypes are not needed for the argument of a predicate, since the necessary type context can be created with the body of the predicate,

(iii) type annotations should be used to simplify inferred types, and

(iv) types should be declared to belong to `TYPE1` whenever appropriate, since the inhabitation property is built into the type checker.

The effectiveness of the our approach depends on the type injection rules (see Section 9.4.6, the simplifier and their interaction. Consider the following declarations:

```
p:A -> Bool ; q:A -> Bool ;
S == {p} ;
T == {q} ;
R == {r:S | q r}
```

A term of type `R` will be accepted wherever either `S` or `T` is required, but the mechanism differs. In the first case, `S` is the base type of `R`: this is relationship is captured in the type injection rules. In the second case, both `T` and `R` are characterised by the predicate `q`, resulting in a type correctness condition of the form `q x => q x` which is removed by the simplifier. To benefit from this simplification, `T` and `R` must be characterised by the predicates which are *syntactically* identical and not just semantically equivalent.

6.3.3.3 'The Trouble with Predicate Subtypes'

In their evaluation [LP98] of types in specification languages, Lamport and Paulson point out three potential disadvantages of predicate subtypes.

1. A well-typed expression may contain sub-expressions which are not well-typed out of context, restricting how one can decompose definition. As an example they give the definition

   $$P \triangleq (i - j \geq 0) \Rightarrow (A[i-j] = A[i-j])$$

   where i, j are variables (defined elsewhere, not bound variables of the definition) of type $\mathbb{N}$, A is an array of type $\mathbb{N} \to \mathbb{N}$ and "$-$" has type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. While this definition is well-typed, in the pair of definitions

   $$Q \triangleq (A[i-j] = A[i-j]) \quad P \triangleq (i - j \geq 0) \Rightarrow Q$$

   the first definition Q is not well-typed. In our notation, we could write

   ```
   Q (i:Nat) (j:{(>=) i}) == A (i - j) = A (i - j)
   ```

   which, at the expense of a type declaration, precisely specifies the type checking required to permit this decomposition.

2. The context for the type-correctness of an expression in a programming statement must be established. The following example is given:

   $$\begin{array}{l} \textbf{initially}\ n = 0; s = [\,] \\ \textbf{do true}\ \to\ n := n + 1; s := s \oplus [42] \\ \quad \Box \\ \quad n > 0\ \to\ n := n - 1; s := \mathit{Tail}(s) \end{array}$$

   Lamport and Paulson note that the function *Tail* must be applied to a non-empty list; in the example this follows from the loop invariant $n = \mathit{Len}(s)$ and they conclude that, in general 'the type declaration of the program variables will have to encode an invariant of the program'. The opposite approach is taken here: the loop invariant, which is part of the loop syntax, contributes to the type context (see Section 11.10) for type checking terms in programs. As a result, this example presents no difficulty.

3. The logical inference rules for languages with predicate subtypes are complex.

The first two of these points are resolved in the approach described here, although type annotations may be required. We consider the acceptability of these type annotations to be a pragmatic issue which must be resolved by users.

The rules for type checking in context do complicate the logical inference rules in a language of predicate subtypes. This needs to be set against the simplicity achieved by ensuring that all functions are total. The development of logical inference rules taking account of subtypes is not addressed in this thesis.

6.3.3.4 Completeness of the Type Checker

We note here a small limitation of our type checker. In some circumstances, unprovable type correctness conditions are produced for terms which might be considered to be well-typed. Consider the following example:

```
IF_T (b:Bool) (A:TYPE) (B:TYPE) == IF b THEN A ELSE B END ;
c : Bool ;
AXIOM C == c ;
type_ok == 0 : IF_T c Nat Bool ;
n : IF_T c Nat Bool ;
not_type_ok == n : IF_T c Int Bool
```

The constant `type_ok` requires `0` to type check as `IF_T c Nat Bool`, which causes no difficulty. However, the constant `not_type_ok` effectively requires the type checker to infer that `IF_T c Nat Bool` is a subtype of `IF_T c Int Bool`. This results in an unprovable type condition:

```
OBLIGATION not_type_ok_cc1 ==
  (c => IF_T c Nat Bool = Int) && (not c => IF_T c Nat Bool = Bool)
```

This problem arises because, as noted on page 127, the type injection rules use equality where a subtype relation is required. If this were expressible in our language, the type correctness condition would become:

```
(c => IF_T c Nat Bool SUB_TYPE Int) &&
(not c => IF_T c Nat Bool SUB_TYPE Bool)
```

An alternative approach would be to extend our language to include type assignment and subtype *as terms*. In this case, `t HAS_TYPE T` would be a boolean term, true if `t` is an element of the type `T`. The inference rules for this extended logic would include both the rules for type assignment and the existing logical inference rules. This would allow

the type checker to return a type correctness condition using SUB_TYPE or HAS_TYPE when none of the existing rules allowed it to proceed. We do not consider that the practical effect of this change would be very great.

This problem may not arise in PVS. The syntactic distinction between terms and types prevents the construction of an equivalent of IF_T. The price is the use of parametric theories and overloading. Adding HAS_TYPE and/or SUB_TYPE to our language would allow PVS's type judgements to be stated as axioms, distinguished only by the type checker's special use of them. Type judgements in PVS appear to be special extensions which do not form part of the logical language itself.

Since Veritas integrates type checking and logical inference, it is not expected that this problem would arise in Veritas. However, interactive type checking is likely to be required in cases equivalent to the one shown above. Indeed, as noted in Section 6.2.2, Veritas include a term $\sigma \subseteq \sigma'$ expressing the subtype property, equivalent to

$$(\forall \nu : \sigma \cdot \exists \nu' : \sigma' \cdot \nu = \nu')$$

Type membership can be characterised in a similar way: the theorem required to show that τ has type σ (possibly different from the inferred type) is

$$(\exists \nu : \sigma \cdot \tau = \nu)$$

If Veritas' rule for the formation of equality is used, type correctness conditions of these forms could be generated by the type checker. The price to pay would be weaker well-formation checking, since $3 = false$ would become a well-formed term.

Part II

Language Specification

Chapter 7

Overview of the Language Specification

7.1 Objectives of the Language Specification

The language specification covers the abstract syntax, well-formation rules and the 'proof semantics', showing how the proof obligations for type correctness and refinement are generated. The objectives of the specification are to:

(i) provide a rigorous definition of the language,

(ii) specify the essential functions of the prototype tool.

During the development of the language, ideas have been examined and refined using a combination of small examples, formal specification and tool development. All have been found to be useful.

The specification does not exhibit a model of language, which could be used to prove soundness properties. For the term language, a set-theoretic model could be used; for the statements a structured operational semantics or denotational semantics would also be appropriate.

7.2 Language Components

The abstract syntax of the language contains the following principal categories:

$\mathcal{I}$	identifers
$\mathcal{T}$	terms, including types and predicates
$\mathcal{D}$	declarations, including data types.
$\mathcal{S}$	statements, including refinement trees.
$\mathcal{U}$	units, including theories and refinement texts.

7.3 Structure of the Specification

The structure of the specification is strongly influenced by the treatment of variables and substitution. Variables may stand either for terms or for statements. In particular, quantifiers and λ-abstractions introduce variables standing for terms, while procedure declarations introduce statement variables.

Substitution of variables is widely used in the specification of the language, for example to explain the meaning of assignment statements. Care must be taken to avoid variable capture: the common solution – renaming variable during substitution – turns out to be complex (an earlier version of this work [Mar95] used three variants of substitution), in particular because the statement defining a procedure can contain both term variables and statement variables.

7.3.1 Depth Indices

To avoid these complexities, we use De-Bruijn or 'depth' indices [Bru72] instead of identifiers to represent variables. As well as avoiding variable capture, the use of indices makes α-conversion unnecessary. A depth index is a natural number, written $\underline{0}$, $\underline{1}$, $\underline{2}$ ... or $\underline{i}$. Using depth indices to represent the term $(\lambda x : \mathbb{N}.(\lambda y : \mathbb{N}.x + y))$, the variable y in the subterm $x + y$ is represented by $\underline{0}$ since y is introduced by the innermost abstraction and the variable x by $\underline{1}$ since x is introduced by the next abstraction outwards. The complete representation of the term is then $(\lambda\mathbb{N}.(\lambda\mathbb{N}.\underline{1} + \underline{0}))$.

7.3.2 Translating from Identifiers to Indices

There are two variants of each syntactic category. In the first variant, which models the concrete syntax, variables are represented by identifiers (from the set $\mathcal{I}$). To distinguish this variant, a subscript x is used: $\mathcal{T}_x$, $\mathcal{D}_x$. In the second variant $\mathcal{T}_i$, $\mathcal{D}_i$ ... variables are represented by depth indices. Variables can occur free in both terms and statements. Variables are only introduced as bound variables, in syntactic forms such as the λ-abstraction or the procedure declaration. As a result, the terms used in declarations and the statements used in a refinement text are *closed*, containing bound variables but no free variables.

The language specification has two steps:

(i) a translation or *encoding* from a syntax with identifiers to a syntax based on indices.

(ii) specification of well-formation, type checking and refinement checking on terms of the abstract syntax using indices.

The representation of constants also differs between the two variants of the abstract syntax. In the syntax with identifiers, directly visible constants are represented by a single identifier while indirectly visible constants have a theory identifier prefix. In the abstract syntax with indices, all constants are represented by two identifiers.

In the syntax with identifiers, directly visible constants have the same syntax as variables. The encoding step checks the visibility of identifiers and distinguishes variables from constants; in the syntax with indices variables and constants have distinct forms. Moreover, the distinction between direct and indirect visibility exists only in the syntax with identifiers.

7.4 Notation

The notation used for the formal parts of the specification is based on simply typed set theory. The notation is similar to Z [Spi89] but with simplifications and less formality; schemas are not used. A boolean type is used.

The same notation is used for the object language and the meta-language. We rely on the context to make the necessary distinction.

Chapter 8

Environments, Declarations and Theories

In this chapter, declarations and theories are described. Each declaration is checked in the context of the preceeding declarations. This context is described by 'environments' which are introduced first. Only the simpler declarations are described in this chapter; description of the datatype declaration is postponed to Chapter 10.

8.1 Environments

Two environment structures are required:

The visibility environment provides the context for the translation from the abstract syntax with identifiers to the variant with indices. The environment allows:

1. identifiers representing variables to be converted to depth indices, and
2. the visibility of constants to be checked, determining the theory in which a directly visible constant is declared.

The type environment provides the context for checking the well-formation and the correctness of a refinement text. The environment allows:

1. the type of a variable to be determined
2. the type of a constant to be determined and, when applicable, its definition as well.

Both of these environments are outlined below. Full details are built up in succeeding chapters.

8.1.1 The Visibility Environment

The directly visible constant identifiers (from $\mathcal{I}$) are mapped to the identifier of the theory in which they are declared by the function $\delta \in \mathit{VisDir} == \mathcal{I} \nrightarrow \mathcal{I}$. The set of directly visible constant identifiers is $\mathbf{dom}\,\delta$.

The function $\iota \in \mathit{VisInd} == \mathcal{I} \nrightarrow \mathbb{P}\mathcal{I}$ maps a theory identifier to the set of constants which are indirectly visible within the theory. The set of visible theories is therefore $\mathbf{dom}\,\iota$, while the constants which are indirectly visible are $\bigcup\{\, l \in \mathbf{dom}\,\iota \cdot \{x : \iota\, l \cdot l.x\}\,\}$.

The function $\nu \in \mathit{VisVar} == \mathcal{I} \nrightarrow \mathbb{N}$ maps a variable identifier to a depth index. The set of visible variables is $\mathbf{dom}\,\nu$. The complete environment combines these components.

$$\rho \in \mathit{VisEnv} == \mathit{VisInd} \times \mathit{VisDir} \times \mathit{VisVar}$$

The following properties are maintained for $(\iota, \delta, \nu) \in \mathit{VisEnv}$.

1. The sets of directly visible constants and visible variables are disjoint:

   $$\mathbf{dom}\,\delta \cap \mathbf{dom}\,\nu = \emptyset$$

   The same identifier may be used to name a theory as well as a constant or a variable. This is permitted since a theory identifier can only be used as the prefix of a constant name.

2. No theory is both directly and indirectly visible:

   $$\forall c : \mathbf{dom}\,\delta \cdot \delta\ c \notin \mathbf{dom}\,\nu$$

   This rule ensures that there is a unique way to refer to a constant.

Adding a Variable to the Environment We define an operation $_ + _$ to add a new variable identifer to the visibility environment $\rho : \mathit{VisEnv}$:

$$(\iota, \delta, \nu) + x \quad = \quad (\iota, \{x\} \lhd \delta, ((\lambda n : \mathbb{N} \cdot n + 1) \circ \nu) \oplus \{x \mapsto 0\})$$

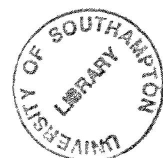
The new variable x has index zero; the indices of other variables are increased by one. The new variable masks other uses of the same identifier.

8.1.2 The Type Environment

The type of a variable is a term, the 'type' of a statement variable is a statement[1]. Constants may have a definition as well as a type.

$$\begin{aligned} \mathit{VType} &== \mathit{Var}\ \mathcal{T}_i \mid \mathit{Proc}\ \mathcal{S}_i \\ \mathit{CType} &== \mathit{Decl}\ \mathcal{T}_i \mid \mathit{Defn}\ \mathcal{T}_i\ \mathcal{T}_i \end{aligned}$$

[1]In Chapter 11 it is described that each statement variable stands for a statement.


UNIVERSITY OF SOUTHAMPTON LIBRARY

The function $\kappa \in ConTyp == \mathcal{I} \times \mathcal{I} \to CType$ maps constant identifiers – always prefixed with the theory name – to constant types. The sequence $\nu \in VarTyp == \mathbf{seq}\, VType$ contains the types of the variables: the type of the most recently declared variable $\underline{0}$ is at the head of the sequence. The type environment $\rho \in TypEnv == ConTyp \times VarTyp$ combines the variable and constant types.

Adding Variables to the Environment We define an operation $_+_$ to add a new variable to the type environment $\rho : TypEnv$:

$$\begin{aligned}(\kappa, \nu) + t &= (\kappa, \langle Var\ t\rangle \frown \nu)\\ (\kappa, \nu) + s &= (\kappa, \langle Proc\ s\rangle \frown \nu)\end{aligned}$$

The second case applies to statement variables X, the first case to 'ordinary' variables x. We rely on the context to resolve the overloading of the $_+_$ operation.

Environment Lookup The $_!!_$ function returns the type of a variable or constant. The function is overloaded[2] for the two sorts of variables.

$$\begin{aligned}(\kappa, \nu) !! \underline{i} &= t^{+i} && \textbf{where } \nu !! i = Var\ t\\ (\kappa, \nu) !! \underline{I} &= s^{+I} && \textbf{where } \nu !! I = Proc\ s\\ (\kappa, \nu) !! l.c &= t && \textbf{where } \kappa\ l.c = Decl\ t\\ (\kappa, \nu) !! l.c &= t && \textbf{where } \kappa\ l.c = Defn\ t'\ t\end{aligned}$$

Note that the type of a variable may itself contain variables. The function $_^{+}$ is used to adjust the depth indices so that the context is correct – see page 121.

8.2 Simple Declarations

Constants A declaration can introduce a constant in two ways: either by giving a name to a defining term ($x == A$), or by giving a name to a term with a type but without a definition ($x : A$). The second form of declaration assumes that a term with the specified type exists; if this is not the case the declaration introduces an inconsistency.

Axioms and Theorems A declaration can introduce a formula which is either assumed to be true (an axiom) or can, it is conjectured, be proved to be true (a theorem).

[2]Note that the notation is further overloaded: the ith elment of a list l is $l !! i$.

$$
\begin{array}{lcl}
[\![x == t]\!]^{l}_{\iota,\delta} & = & l.x == [\![t]\!]_{\iota,\delta,\emptyset} \textbf{ provided } x \notin \textbf{dom}\,\delta \\
[\![x : t]\!]^{l}_{\iota,\delta} & = & l.x : [\![t]\!]_{\iota,\delta,\emptyset} \textbf{ provided } x \notin \textbf{dom}\,\delta \\
[\![\textbf{formula}\, x == t]\!]^{l}_{\iota,\delta} & = & \textbf{formula } [\![t]\!]_{\iota,\delta,\emptyset} \\
[\![d_1 ; d_2]\!]^{l}_{\iota,\delta} & = & [\![d_1]\!]^{l}_{\iota,\delta}; [\![d_2]\!]^{l}_{\iota,\delta'} \textbf{ where } \delta' = decl_id_{\iota,\delta}\, d_1 \\
 & & \\
decl_id_{\iota,\delta}\, l\ (x : t) & = & \delta \oplus (x, l) \\
decl_id_{\iota,\delta}\, l\ (x == t) & = & \delta \oplus (x, l) \\
decl_id_{\iota,\delta}\, l\ \textbf{formula}\, x == t & = & \delta \\
decl_id_{\iota,\delta}\, l\ (d_1 ; d_2) & = & decl_id_{\iota,\delta'}\, l\ d_2 \textbf{ where } \delta' = decl_id_{\iota,\delta}\, l\ d_1
\end{array}
$$

Figure 8.1: Encoding of Declarations

8.2.1 Abstract Syntax

The abstract syntax of declarations with identifiers is:

$$\mathcal{D}_x = \mathcal{I} == \mathcal{T}_x \mid \mathcal{I} : \mathcal{T}_x \mid \textbf{formula}\, x == \mathcal{T}_x \mid \mathcal{D}_x ; \mathcal{D}_x$$

The abstract syntax with indices has exactly the same form, except that the axiom and theorem identifiers are discarded[3]. The theory identifier is made explicit.

$$\mathcal{D}_i = (\mathcal{I}, \mathcal{I}) == \mathcal{T}_i \mid (\mathcal{I}, \mathcal{I}) : \mathcal{T}_i \mid \textbf{formula}\, \mathcal{T}_i \mid \mathcal{D}_i ; \mathcal{D}_i$$

8.2.2 Visibility in Declarations

The rules for the visibility of identifiers in declarations are:

(i) the identifiers introduced in a declaration must not already be directly visible, preventing redeclaration of constants, and

(ii) the identifier introduced in a declaration is directly visible in succeeding declarations.

Redeclaration of constants is prevented to avoid the meaning of terms varying with context. The encoding of declarations is described by the functions:

$$
\begin{array}{l}
[\![_]\!]^{-}_{-} : \mathit{VisInd} \times \mathit{VisDir} \mathrel{+\!\!\!\!\rightarrow} \mathcal{I} \mathrel{+\!\!\!\!\rightarrow} \mathcal{D}_x \mathrel{+\!\!\!\!\rightarrow} \mathcal{D}_i \\
decl_id_ : \mathit{VisInd} \times \mathit{VisDir} \rightarrow \mathcal{I} \rightarrow \mathcal{D}_x \rightarrow \mathit{VisDir}
\end{array}
$$

[3] Axioms and theorems cannot be refered to in other declarations; their identifiers can therefore be taken from a separate namespace. We omit the specification of the uniqueness of each theorem name, since this is not required for type checking.

$$
\begin{array}{lcl}
decl_ok_{\kappa}\ l.x == t & = & type_ok_{\kappa,\emptyset}\ t \\
decl_ok_{\kappa}\ l.x : t & = & type_ok_{\kappa,\emptyset}\ t \wedge \textbf{kind_or_type}(type_inf_{\kappa,\emptyset}\ t) \\
decl_ok_{\kappa}\ \textbf{formula}\ t & = & type_ok_{\kappa,\emptyset}\ t \wedge \textbf{is_boolfun_type}(type_inf_{\kappa,\emptyset}\ t) \\
decl_ok_{\kappa}(d_1; d_2) & = & decl_ok_{\kappa}\ d_1 \wedge decl_ok_{\kappa'}\ d_2\ \textbf{where}\ \kappa' = decl_type_{\kappa}\ d_1
\end{array}
$$

$$
\begin{array}{lcl}
decl_type_{\kappa}\ l.x : t & = & \kappa \oplus ((l, x), Decl\ t) \\
decl_type_{\kappa}\ l.x == t & = & \kappa \oplus ((l, x), Defn\ t\ \ type_inf_{\kappa,\emptyset}\ t) \\
decl_type_{\kappa}\ \textbf{formula}\ t & = & \kappa \\
decl_type_{\kappa}(d_1; d_2) & = & decl_type_{\kappa'}\ d_2\ \textbf{where}\ \kappa' = decl_type_{\kappa}\ d_1
\end{array}
$$

Figure 8.2: Well-Formation of Declarations

where $[\![d]\!]^{l}_{\iota,\delta}$ is the encoding of the declaration d, in the theory l and in the environment ι, δ. The function *decl_id* describes how declarations add identifiers to the environment. The encoding of a term t is $[\![t]\!]_{\rho}$ – see Section 9.3. The specification of these functions is shown in Figure 8.1.

8.2.3 Well-formation of Declarations

The well-formation rules for a declaration $d \in \mathcal{D}_i$ are:

(i) the defining term t of a definition $x == t$ must be well-typed,

(ii) the type of the term t used in a declaration $x : t$ must be $*$ or $\Box$; this is the same rule as is applied to bound variable declarations in λ-abstractions, and

(iii) a term used in an axiom or theorem must be a boolean, or a function to a boolean.

The final rule allows an axiom or theorem declaration to have arguments, forming an 'axiom scheme'. In effect, there is universal quantification over each argument. The well-formation of declarations is described by the function *decl_ok* and *decl_type*, which describes how each declaration is added to the environment:

$$
\begin{array}{l}
decl_ok_ : ConTyp \nrightarrow \mathcal{D}_i \nrightarrow \mathbb{B} \\
decl_type : ConTyp \rightarrow \mathcal{D}_i \rightarrow ConTyp
\end{array}
$$

The specifications are in Figure 8.2. The functions $type_ok_{\rho}$ and $type_inf_{\rho}$ check the well-formation of a term and return its type respectively: see Section 9.4. The functions **kind_or_type** and **is_boolfun_type** are declared in Section 9.4.7.

8.3 Theories

Declarations are grouped into theories which can be given names. In this section, a very simple treatment of theories is given, in which:

(i) all declarations in a theory are exported.

(ii) a theory may import other theories either *directly* (indicated by the keyword SEE) or *indirectly*.

If a theory is imported directly, then all the constants from the imported theory are directly visible; otherwise all the constants must be prefixed with the theory identifier. This is a simple approach, providing some control over the name space for constants. A more elaborate approach, similar to that used in the language Haskell [PE97], would allow a subset of the declarations in a theory to be exported. The most important aspect of our treatment is that the distinction between direct and indirect visibility affects only the translation between the two versions of the abstract syntax. As a result, elaboration of the rules for import/export would not change the rules for type checking.

8.3.1 Abstract Syntax

Theories belong to the category $\mathcal{U}$ of units. The elements of a theory are its name, the names of directly imported theories and a declaration. A second form of unit is introduced in Chapter 12. In the syntax with indices, the theory identifier and the list of directly imported theories are not required.

$$\mathcal{U}_x = \mathbf{theory}\,\mathcal{I}\ \mathbf{seq}\,\mathcal{I}\ \mathcal{D}_x \mid \ldots \qquad \mathcal{U}_i = \mathbf{theory}\,\mathcal{D}_i \mid \ldots$$

Texts A sequence of units form a text: $Text == \mathbf{seq}\,\mathcal{U}$.

8.3.2 Visibility in Theories

The rules for the visibility of identifiers in theories are:

(i) the identifier used to name the theory must be different from existing theories,

(ii) the list of theory names to be directly imported must be disjoint and all must be visible, and

(iii) the identifiers of constants declared in the directly imported theories must all be different.

$$
\begin{array}{lcl}
[\![\mathbf{theory}\, l\ ls\ d]\!]_\iota & = & \mathbf{theory}[\![d]\!]^l_{\iota',\delta} \\
\multicolumn{3}{l}{\qquad \mathbf{provided}\ l \notin \mathbf{dom}\,\iota \wedge ls \in \mathbf{iseq} \wedge} \\
\multicolumn{3}{l}{\qquad\qquad \mathit{disjoint}\ \langle l \in ls \cdot \iota\ l\rangle \wedge (\forall l : ls \cdot l \in \mathbf{dom}\,\iota)} \\
\multicolumn{3}{l}{\qquad \mathbf{where}\ \delta = \bigcup\{\ l \in ls \cdot (\lambda x : \iota\ l \cdot l)\ \}} \\
\multicolumn{3}{l}{\qquad\qquad \iota' = ls \lhd \iota}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\langle\rangle]\!]_\iota & = & \langle\rangle \\
[\![\langle u\rangle \frown us]\!]_\iota & = & \langle [\![u]\!]_\iota \rangle \frown [\![us]\!]_{\iota'}\ \mathbf{where}\ \iota' = \mathit{unit_id}_\iota\, u
\end{array}
$$

$$
\mathit{unit_id}_\iota\, \mathbf{theory}\, l\ is\ d = \iota \cup \{(l, \mathbf{dom}(\delta' \rhd \{l\}))\}\ \mathbf{where}\ \delta' = \mathit{decl_id}_{\iota,\emptyset}\, d
$$

Figure 8.3: Unit and Text Encoding

$$
\begin{array}{lcl}
\mathit{unit_ok}_\kappa\, \mathbf{theory}\, d & = & \mathit{decl_ok}_\kappa\, d \\
 & & \\
\mathit{units_ok}_\emptyset \langle\rangle & = & \mathbf{t} \\
\mathit{units_ok}_\kappa \langle u\rangle \frown us & = & \mathit{unit_ok}_\kappa\, u \wedge \mathit{units_ok}_{\kappa'}\, us \\
 & & \mathbf{where}\ \kappa' = \mathit{unit_type}_\kappa\, u \\
 & & \\
\mathit{unit_type}_\kappa\, \mathbf{theory}\, d & = & \mathit{decl_type}_\kappa\, d
\end{array}
$$

Figure 8.4: Well-fromation of Units and Texts

The encoding of a unit u is $[\![u]\!]_\iota$, where $\iota : \mathit{VisInd}$ is the environment. Similarly, the encoding of a text us is $[\![us]\!]_\iota$. The function $\mathit{unit_id} : \mathit{VisInd} \rightarrow \mathcal{U}_x \rightarrow \mathit{VisInd}$ describes how a unit add identifiers to the environment. These functions are specified in Figure 8.3.

8.3.3 Well-formation of Units

A theory unit u is well-formed $\mathit{unit_ok}_\kappa$ if the declaration it contains is well-formed in environment $\kappa : \mathit{ConTyp}$. The well-formation of a text follows in the obvious way; the specifications are shown in Figure 8.4. Note that the empty text is well-formed in the empty environment.

Chapter 9

Terms

The language of terms is based on the language λC described in Chapter 2. We start by introducing and motivating some extensions to λC. Sections 9.2 and 9.3 cover the abstract syntax of the term language and the rules governing the visibility of identifiers. Section 9.4 describes an algorithm for type checking terms. The final section enhances the algorithm to report type errors.

9.1 Using the λ-Cube

The λC language is extended with logical operators, quantifiers, new terms for conditional, equality, subtype and choice[1]. Three methods of extending λC are available:

(i) conservative extension, where new terms are defined from existing terms,

(ii) declaration, where new terms are declared but not defined; axioms are needed to specify their properties, and

(iii) extending the syntax of λC with new forms, with type assignment rules following the pattern of Figure 2.1 on page 11.

A combination of these methods is used below and some alternatives are evaluated.

9.1.1 Extensions to λC

Figure 9.1 shows the type assignment rules for the following new terms.

Conditional A conditional term is introduced.

[1]Some further terms associated with datatypes and recursion are added in Chapters 10.

(Conditional) $\dfrac{\Gamma \vdash p : \mathbb{B} \quad \Gamma \vdash a : A \quad \Gamma \vdash a' : A}{\Gamma \vdash \mathbf{cond}\, p\ a\ a' : A}$

(Forall) $\dfrac{\Gamma \vdash p : (\Pi x : A.\mathbb{B})}{\Gamma \vdash \forall p : \mathbb{B}}$

(Exists) $\dfrac{\Gamma \vdash p : (\Pi x : A.\mathbb{B})}{\Gamma \vdash \exists p : \mathbb{B}}$

(Equality) $\dfrac{\Gamma \vdash a : A \quad \Gamma \vdash a' : A}{\Gamma \vdash a = a' : \mathbb{B}}$

(Conversion rule') $\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : \{\Box, *\}}{\Gamma \vdash A : B'}$ **provided** $B = B'$

(Subtype) $\dfrac{\Gamma \vdash p_A : (\Pi x : A.\mathbb{B}) \quad \Gamma \vdash A : s}{\Gamma \vdash \{\ p_A\ \} : s}$ **provided** $s \in \{\Box, *\}$

(Injection) $\dfrac{\Gamma \vdash a : A}{\Gamma \vdash a : \{\ p_A\ \}}$ **provided** $p_A\ a$

(Choice) $\dfrac{\Gamma \vdash p : (\Pi a : A.\mathbb{B}) \quad \Gamma \vdash A : \{\Box, *\}}{\Gamma \vdash \epsilon p : A}$ **provided** $(\exists a : A.\mathbf{t})$

Figure 9.1: Type Assignment for λC Extensions

Forall and Exists Universal and existential quantifier terms $\forall$ and $\exists$ are introduced. We use conventional syntax for the quantifiers, writing $(\forall x : A.p)$ instead of $\forall(\lambda x : A.p)$.

Equality We introduce an equality operator term $_ = _$. In addition, the conversion rule of Figure 2.1 is changed to allow a type to be replaced by an equal type, rather than only by one that is β-convertible. Unlike the requirement of β-convertibility, the equality proviso $B = B'$ is not decidable.

Subtypes The subtype term $\{\ p_A\ \}$ has two type assignment rules: the second rules show how an element of a subtype is formed. The proviso $p_A\ a$ states that the term a must satisfy the predicate p_A which characterises the subtype. We write $\{\ t : T \mid p\ \}$ as an alternative to $\{\ (\lambda t : T \cdot p)\ \}$.

In Section 3.4.1, the sort of inhabited types is declared:

$$\mathbf{type1} = \{\ T : * \mid (\exists T \cdot \mathbf{t})\ \}$$

It is notable that a subtype defined from an inhabited type is not necessarily inhabited. Thus $\{t : T \mid \mathbf{f}\ \}$ has type $*$, even if $T : \mathbf{type1}$.

Choice We introduce a choice term ϵ, which selects a value from a type satisfying a predicate (or an arbitrary term from the type if the predicate cannot be satisfied).

In the syntax ϵ is treated as a quantifier so that $(\epsilon\, x : A.p)$ may be written instead of $\epsilon(\lambda x : A.p)$.

Choosing an arbitrary term from a type could cause inconsistency since not all types are inhabited. As noted on Page 12, there are uninhabited types in λC even without the introduction of subtypes. To avoid this problm, we require the type from which a term is chosen to be inhabited.

9.1.2 Initial Context

The type assignment rules of Figure 9.1 refer to the boolean type $\mathbb{B}$, even though this is not a term. We assume that type assignment is carried out in an initial context which provides this type and the logical operators. The boolean type and constants $\mathbf{t}$ (true) and $\mathbf{f}$ (false) are declared:

$$\mathbb{B} : * \quad \mathbf{t} : \mathbb{B} \quad \mathbf{f} : \mathbb{B}$$

Axioms are required to specify the properties of the boolean type. The propositional operators are defined as follows:

$$\begin{array}{lcl} p \Rightarrow q & \equiv & \mathbf{cond}\, p\ q\ \mathbf{t} \\ \neg p & \equiv & p \Rightarrow \mathbf{f} \\ p \wedge q & \equiv & \neg(p \Rightarrow \neg q) \\ p \vee q & \equiv & \neg p \Rightarrow q \\ p \Leftrightarrow q & \equiv & p \Rightarrow q \wedge q \Rightarrow p \end{array}$$

9.1.3 An Alternative Approach

The extensions described above can be achieved without introducing any new terms except the subtype constructor[2]. One possible definition of the boolean type and associated definitions and declarations are:

$$\begin{array}{rcl} \mathbb{B} & \equiv & * \rightarrow * \rightarrow * \\ \mathbf{t} & \equiv & (\lambda A : *.\lambda B : *.A) \\ \mathbf{f} & \equiv & (\lambda A : *.\lambda B : *.B) \\ \Rightarrow & \equiv & (\lambda p : \mathbb{B}.\lambda q : \mathbb{B}.\lambda A : *.\lambda B : *.p\ (q\ A\ B)\ A) \\ \mathbf{cond} & \equiv & (\lambda A : *.\lambda p : \mathbb{B}.\lambda t : A.\lambda u : A.p\ t\ u) \\ = & : & T : * \rightarrow T \rightarrow T \rightarrow \mathbb{B} \\ \forall & \equiv & (\lambda T : * \cdot \lambda f : (T \rightarrow \mathbb{B}) \cdot f = (\lambda a : A \cdot \mathbf{t})) \\ \exists & \equiv & (\lambda T : * \cdot \lambda f : (T \rightarrow \mathbb{B}) \cdot \neg(\forall a : A \cdot \neg(f\ a))) \\ \epsilon & : & T : \{\, U : * \mid (\exists u : U \cdot \mathbf{t})\, \} \rightarrow (T \rightarrow \mathbb{B}) \rightarrow T \end{array}$$

Axioms would be required to specify the properties of equality and the choice term. As a *practical* way of providing logic, these definition have the disadvantages that the

[2]Even the subtype constructor can be modelled by a declared function.

boolean type is parameterised by arbitrary types and, further, its representation cannot be hidden. Two further difficulties of using these declarations and definitions arise from the functions **cond**, $=$, ϵ, $\forall$ and $\exists$ needing a type argument.

1. To universally quantify the predicate $p : A \rightarrow \mathbb{B}$, where $A : *$, the application $\forall\ A$ must be used.

2. The functions such as $\forall$ and $=$ are polymorphic over elements of the sort $*$ only. Similar quantifiers could be defined for other sorts, such as predicates on kinds, such as $p : * \rightarrow \mathbb{B}$. This approach would lead to a family of quantifiers $\forall_A$, $\forall_*$, $\forall_{*\rightarrow*}$, ...

The first difficulty can be overcome using implicit arguments (see Section 3.2.3) so that the type parameter can be omitted in practice, but the second difficulty remains.

9.1.4 Consistency of Extensions

It is possible that extensions to λC could make the language inconsistent, in the sense that models of the extended language require terms to be identified which are not identified in models of λC. We argue informally that the extensions we have made to λC do not have this effect, because the effect of the extensions can be created in the unextended language, so that the 'extension' adds nothing new.

The equality term $A = B$ adds to the object language the meta-language concept of β-equivalence, with the restriction that only terms of the same type can be compared for equality. The additionals of axioms of equality could make the language inconsistent.

A subtype can be modelled by a type paired with a predicate: the subtype $\{p_A\}$ is modelled by (A, p_A). A function on the subtype $f : (\Pi\{p_A\}.B)$ can be modelled by a function $f' : (\Pi A.B)$ which corresponds to f for values of A satisfying p and is arbitrary elsewhere:

$$f' = (\lambda a : A.\, \mathbf{cond}(p\ a)\ (f\ a)\ _)$$

This approach requires expressions involving the application of a function from a subtype to be re-written to make the subtype constraint explicit. For example $f\ a = b$ becomes $p\ a \wedge f'\ a = b$. Thompson show in [Tho92] that in the constructive theory of types subtypes can be modelled by the existential type. In constructive type theory, propositions are equated with types. The proposition (type) $(\exists x : A \cdot B)$ has member (a, p) where $a : A$ and $p : B[a/x]$ is a witness of the fact that proposition B holds for a. As a result, the existential quantifier can be thought of as forming a subset of A. Elements of the subset are accompanied by the proof of membership.

The existence of empty types needs to be taken into account in the logical rules. In particular $(\forall t \cdot p)$ holds for any p if t is empty; conversely $(\exists t \cdot p)$ is always false.

	$\mathcal{T}_x$	$\mathcal{T}_i$
kind sort	$\square$	
type sort	$*$	
variable[3]	x	$\underline{i}$
constant	$l.x$	
application	$\mathcal{T}\ \mathcal{T}$	
abstraction	$\lambda x : \mathcal{T} \cdot \mathcal{T}$	$\lambda \mathcal{T} \cdot \mathcal{T}$
product	$\Pi x : \mathcal{T} \cdot \mathcal{T}$	$\Pi \mathcal{T} \cdot \mathcal{T}$
forall	$\forall \mathcal{T}$	
choice	$\epsilon \mathcal{T}$	
conditional	$\mathbf{cond}\, \mathcal{T}\ \mathcal{T}\ \mathcal{T}$	
equality	$\mathcal{T} = \mathcal{T}$	
subtype	$\{\ \mathcal{T}\ \}$	
annotation	$\mathcal{T}$ **typed** $\mathcal{T}$	

Notes: 1. *Where $\mathcal{T}_i$ is not shown it has the same form as $\mathcal{T}_x$.*
2. *The annotation term is introduced in Section 9.4.2.*

Figure 9.2: Abstract Syntax of Terms

9.2 Abstract Syntax

Figure 9.2 shows the abstract syntax of the terms language. Compared to the forms described in Section 3.1, we omit the existential choice since its treatment follows from that of the universal extension. The numerals are also omitted. Two versions of the abstract syntax are shown: $\mathcal{T}_x$ has identifiers and resembles the concrete syntax, while $\mathcal{T}_i$ has indices, as described in Section 7.3.1. In the figure, x is an identifier, the index i is a natural number and t, u, f, a are terms.

Free Variables Let t be $i\ (\lambda j \cdot k\ \underline{0})$ where i, j, k are indices, greater than zero. The free indices of t are i, j and $k - 1$; the index $\underline{0}$ is not free since it is bound by the abstraction. The free variables of t are denoted by $\Phi\, t$, satisfying:

$$\begin{array}{lcl} \Phi(\lambda t \cdot u) & = & \Phi\, t \cup \{\ i - 1 \mid i \in \Phi\, u \wedge i > 0\ \} \\ \Phi\, \underline{i} & = & \{i\} \end{array}$$

Φ is defined similarly for other terms.

[3]In $\mathcal{T}_x$, a constant may represented by x or $l.x$; a variables is represented by x. In $\mathcal{T}_i$, all constants are represented as $l.x$.

Adjusting Indices The index of a variable depends on the scope in which the variable occurs. When the scope of a term is changed, indices need to be adjusted. We denote the term t with depth indices incremented by one as t^+. The increment operation $_^+$ only increments free indices, so that for the term $t = (\lambda \mathbb{N} \cdot \underline{1} + \underline{0})$, the term t^+ is $t = (\lambda \mathbb{N} \cdot \underline{2} + \underline{0})$. We use the notation t^{+n} for the increment applied n times. The inverse to increment is decrement, written t^- and t^{-n}. A term t cannot be decremented if $\underline{0}$ is free in t.

Substitution The term t with the variable at index i replaced by term u is written $t[i := u]$. When i is zero, this is abbreviated to $t[u]$. The substitution operator $_[_ := _]$ does not need to rename variables to avoid variable capture, since variables are not used. However, it is necessary to adjust the indices in the substituting term whenever the context changes. The most important parts of the specification are as follows:

$$\begin{array}{lcl}
(\lambda t \cdot u)\ [i := v] & = & (\lambda t\ [i := v] \cdot u[i + 1 := v^+]) \\
\underline{j}\ [i := v] & = & \underline{j} \quad \textbf{if } j \neq i \\
 & = & v \quad \textbf{if } j = i
\end{array}$$

β-reduction (see page 129) provides an example of the use of substitution. The reduction of a (curried) function of two arguments is:

$$\begin{array}{lcl}
(\lambda A.(\lambda B \cdot c))\ a\ b & \rightsquigarrow & ((\lambda B.c)[0 := a^+])^-\ b \\
 & = & (\lambda B[0 := a^+] \cdot c[1 := a^{+2}])^-\ b \\
 & = & ((\lambda B[0 := a^+] \cdot c[1 := a^{+2}])\ b^+)^- \\
 & \rightsquigarrow & (c[1 := a^{+2}][0 := b^{+2}])^{-2}
\end{array}$$

9.3 Visibility of Identifiers

A term is well-formed only if all the identifiers used in the term refer to a *visible* constant or variable. In this section, the visibility rules for terms are specified. The visibility rules are defined by the 'encoding' of terms from $\mathcal{T}_x$ as terms in $\mathcal{T}_i$. Since $\mathcal{T}_i$ does not contain terms refering to identifiers which are not visible, the side conditions to the rules specifying this transformation determine the visibility of variables in the $\mathcal{T}_x$ syntax.

The encoding for $t \in \mathcal{T}_x$ is $[\![t]\!]_\rho$ where $\rho \in \mathit{VisEnv}$ is the environment of visible constants and variable, described on page 111. The encoding function is specified by (giving only the case where $\mathcal{T}_x$ differs from $\mathcal{T}_i$):

$$\begin{array}{lcl}
[\![x]\!]_{\iota,\delta,\nu} & = & \underline{\nu\ x} \quad \textbf{provided } x \in \textbf{dom}\, \nu \\
[\![x]\!]_{\iota,\delta,\nu} & = & l.x \quad \textbf{provided } x \in \textbf{dom }\ \delta \wedge \delta\ x = l \\
[\![l.x]\!]_{\iota,\delta,\nu} & = & l.x \quad \textbf{provided } l \in \textbf{dom}\, \iota \wedge x \in \iota\ l \\
[\![\lambda x : t \cdot u]\!]_\rho & = & \lambda [\![t]\!]_\rho \cdot [\![u]\!]_{\rho + x} \\
[\![\Pi x : t \cdot u]\!]_\rho & = & \Pi [\![t]\!]_\rho \cdot [\![u]\!]_{\rho + x}
\end{array}$$

9.4 A Type Checking Algorithm

In this section, we specify an algorithm for non-interactive typechecking: terms are parsed and typechecked, resulting in type error messages and lemmas, called *type correction conditions*, which must be proved to show that the type determined is in fact correct. Type errors are considered in more detail in Section 9.5.

9.4.1 Overview

The basis of the typechecker is forward inference using the rules given in Figure 2.1, with the additions of Section 9.1. To assign a type to a term, the sub-terms are assigned types and then one of the rules is chosen to assign a type by combining the type of the sub-terms. However, the use of forward inference is not sufficient, because of the following difficulties.

Context The 'start' and 'weakening' rules which axiomatise the behaviour of contexts cannot automatically be applied in a forward direction, unless the type of an identifier is known.

Subtype Injection The subtype injection rule does not determine the subtype to be used.

Conversion The conversion rule allows the type assigned to a term to be replaced by any equal term, but does not determine this term.

Side Conditions of Type Assignment The subtype injection rule has a side condition, requiring that a term injected into a subtype satisfies the characteristic predicate of the subtype. Since these conditions are not necessarily decidable, it is not possible for the type checker to ensure that the condition holds. The side-condition of the conversion rule is similar.

We observe that some rules have two occurrences of the same type. For example, in the rule for the formation of an application, the domain type of a function is both explicit in the syntax and also inferred as the type of the function argument. As a result, when forward inference is used to determine the type of the function term, the type of the argument is also determined: backward inference is used to check that the argument has the required type. The solution to each of the difficulties described above is as follows.

Context The type of an identifier is determined from the type environment, described in Section 8.1.2.

Subtype Injection The first step is to infer the type of each term without using the rule for subtype injection. The type inferred is uniquely determined and will

include a subtype only if this is explicit, for example as the type of a bound variable. The functions *type_ok* and *type_inf*, specified below, describe this step.

When a term with redundant type information is encountered, both the required and an actual types of one of the subterms will have been determined. Backward inference of the type assignment rules is then used to determine if the required type is valid for the subterm. The function *inject_type* specifies the backward inference step. The rule for subtype injection may be used to inject a term into a subtype, thus this step is called *type injection*.

Conversion The type checker uses rewrite rules, including β-conversion, to convert terms to a normal form, which is 'partial'. The partial normal form is used when two types are compared. The type checker also converts a term to partial normal form before a type is inferred. Both these steps implicitly apply the conversion rule. The rewrite rules are described in Section 9.4.8.

If there are axioms of equality other than β-reduction, the strong normalisation result of Section 2.1.5 no longer holds. Terms with the same partial normal form are equal, but other terms may also be equal. This is the reason for describing the 'normal form' as partial.

Side Conditions of Type Assignment Since the type checker cannot automatically prove the side conditions which arise in type inference and type injection, type correctness conditions are returned. Type correctness conditions arise from type injection, specified by the function *inject_type*. The type correctness conditions from all the subterms of a term and must also be included in the correct context: the function *tcc* describes this.

9.4.2 Type Annotations

We add to the syntax a mechanism for directing the type checker explicitly to inject a term into a given type. It is convenient to treat this 'type annotation' as a term, with abstract syntax:

$$\mathcal{T} ::= \ldots \mid \mathcal{T} \ \textbf{typed} \ \mathcal{T}$$

9.4.3 Type Checking

The predicate $type_ok : TypEnv \to \mathcal{T} \to \mathbb{B}$ describes how the type checker determines a type can be assigned to a term: $type_ok_{\rho}\, t$ is true whenever the type checker can assign a type to term t in environment ρ. The predicate is defined by the equations shown in Figure 9.3. The definition makes use of the auxiliary functions **is_product**, **domain_of**, **kind_or_type** and **is_pred_type** defined in Section 9.4.7. The definition also uses the function *type_inf*, defined in Section 9.4.4, which gives the type assigned to the term.

$$
\begin{array}{lcl}
type_ok_\rho\, * & = & \mathrm{t} \\
type_ok_\rho\, \underline{i} & = & \mathrm{t} \\
type_ok_\rho\, l.x & = & \mathrm{t} \\
type_ok_\rho(f\ a) & = & type_ok_\rho\, f \wedge type_ok_\rho\, a \wedge \\
 & & \quad \mathbf{is_product}(type_inf_\rho\, f) \\
type_ok_\rho(\lambda t \cdot u) & = & type_ok_\rho\, t \wedge \mathbf{kind_or_type}(type_inf_\rho\, t) \wedge \\
 & & \quad type_ok_{\rho+t}\, u \\
type_ok_\rho(\Pi t \cdot u) & = & type_ok_\rho\, t \wedge \mathbf{kind_or_type}(type_inf_\rho\, t) \wedge \\
 & & \quad type_ok_{\rho+t}\, u \wedge \mathbf{kind_or_type}(type_inf_{\rho+t}\, u) \\
type_ok_\rho\{t\} & = & type_ok_\rho\, t \wedge \mathbf{is_pred_type}(type_inf_\rho\, t) \\
type_ok_\rho(\forall f) & = & type_ok_\rho\, f \wedge \mathbf{is_pred_type}(type_inf_\rho\, f) \\
type_ok_\rho(\epsilon\, f) & = & type_ok_\rho\, f \wedge \mathbf{is_pred_type}(type_inf_\rho\, f) \\
type_ok_\rho(\mathbf{cond}\, p\ t\ u) & = & type_ok_\rho\, p \wedge type_inf_\rho\, p \equiv \mathbb{B} \wedge \\
 & & \quad type_ok_\rho\, t \wedge type_ok_\rho\, u \\
type_ok_\rho(t = u) & = & type_ok_\rho\, t \wedge type_ok_\rho\, u \\
type_ok_\rho(t\ \mathbf{typed}\ u) & = & type_ok_\rho\, t \wedge type_ok_\rho\, u \wedge \mathbf{kind_or_type}(type_inf_\rho\, u)
\end{array}
$$

Figure 9.3: Specification of Type Checking

9.4.4 Type Inference

The function $type_inf : TypEnv \rightarrow \mathcal{T} \rightarrow \mathcal{T}$ describes how the type checker determines the type assigned to a term t in environment ρ, given that $type_ok_\rho\, t$ is true. The function is defined by the equations shown in Figure 9.4. The definition makes use of the auxiliary functions **domain_of**, **range_of** and **base_type_of** defined in Section 9.4.7.

9.4.5 Type Correctness Conditions

The function $tcc : TypEnv \rightarrow \mathcal{T} \rightarrow \mathcal{T}$ describes the type correctness conditions determined by the type checker. The type correctness condition $tcc_\rho\, t$ must be proved to show that the term t is well typed in environment ρ, given that $type_ok_\rho\, t$. The function is defined by the equations shown in Figure 9.5. The definition makes use of the auxiliary function **domain_of** defined in Section 9.4.7 and the function *inject_type* defined in Section 9.4.6. The function *type_pred* is defined below.

9.4.5.1 Context

In a number of cases, the type correctness conditions specified by *tcc* have a 'context hypothesis'. These hypotheses are derived either from the logical structure of the term or from type information.

$$
\begin{array}{lcl}
\mathit{type_inf}_{\rho}\, * & = & \Box \\
\mathit{type_inf}_{\rho}\, \underline{i} & = & \rho \,!!\, \underline{i} \\
\mathit{type_inf}_{\rho}\, l.x & = & \rho \,!!\, l.x \\
\mathit{type_inf}_{\rho}(f\ a) & = & \mathbf{range_of}(\mathit{type_inf}_{\rho}\, f)\ a \\
\mathit{type_inf}_{\rho}(\lambda t \cdot u) & = & \Pi t \cdot \mathit{type_inf}_{\rho+t}\, u \\
\mathit{type_inf}_{\rho}(\Pi t \cdot u) & = & \mathit{type_inf}_{\rho+t}\, u \\
\mathit{type_inf}_{\rho}\{t\} & = & \mathbf{base_type_of}(\mathit{type_inf}_{\rho}(\mathbf{domain_of}(\mathit{type_inf}_{\rho}\, t))) \\
\mathit{type_inf}_{\rho}(\forall f) & = & \mathbb{B} \\
\mathit{type_inf}_{\rho}(\epsilon\, f) & = & \mathbf{domain_of}(\mathit{type_inf}_{\rho}\, f) \\
\mathit{type_inf}_{\rho}(\mathbf{cond}\, p\ t\ u) & = & \mathit{type_inf}_{\rho}\, t \\
\mathit{type_inf}_{\rho}(t = u) & = & \mathbb{B} \\
\mathit{type_inf}_{\rho}(t\ \ \mathbf{typed}\ \ u) & = & u
\end{array}
$$

Figure 9.4: Specification of Type Inference

$$
\begin{array}{lcl}
\mathit{tcc}_{\rho}\, * & = & \mathbf{t} \\
\mathit{tcc}_{\rho}\, \underline{i} & = & \mathbf{t} \\
\mathit{tcc}_{\rho}\, l.x & = & \mathbf{t} \\
\mathit{tcc}_{\rho}(f\ a) & = & \mathit{tcc}_{\rho}\, f \wedge \mathit{tcc}_{\rho}\, a \wedge (\mathit{type_pred}_{\rho}\, a \Rightarrow \\
 & & \qquad \mathit{inject_type}_{\rho}\, a\ (\mathbf{domain_of}(\mathit{type_inf}_{\rho}\, f))) \\
\mathit{tcc}_{\rho}(\lambda t \cdot u) & = & \mathit{tcc}_{\rho}\, t \wedge (\forall t \cdot \mathit{tcc}_{\rho+t}\, u) \\
\mathit{tcc}_{\rho}(\Pi t \cdot u) & = & \mathit{tcc}_{\rho}\, t \wedge (\forall t \cdot \mathit{tcc}_{\rho+t}\, u) \\
\mathit{tcc}_{\rho}\{t\} & = & \mathit{tcc}_{\rho}\, t \\
\mathit{tcc}_{\rho}(\forall f) & = & \mathit{tcc}_{\rho}\, f \\
\mathit{tcc}_{\rho}(\epsilon\, f) & = & \mathit{tcc}_{\rho}\, f \wedge (\exists\, \mathbf{domain_of}(\mathit{type_inf}_{\rho}\, f).\mathbf{t}) \\
\mathit{tcc}_{\rho}(t = u) & = & \mathit{tcc}_{\rho}\, t \wedge \mathit{tcc}_{\rho}\, u \wedge \\
 & & \qquad (\mathit{type_pred}_{\rho}\, u \Rightarrow \mathit{inject_type}_{\rho}\, u\ (\mathit{type_inf}_{\rho}\, t)) \\
\mathit{tcc}_{\rho}(\mathbf{cond}\, p\ t\ u) & = & \mathit{tcc}_{\rho}\, p \wedge (p \Rightarrow \mathit{tcc}_{\rho}\, t) \wedge (\neg p \Rightarrow \mathit{tcc}_{\rho}\, u) \wedge \\
 & & \qquad (\mathit{type_pred}_{\rho}\, u \Rightarrow \mathit{inject_type}_{\rho}\, u\ (\mathit{type_inf}_{\rho}\, t)) \\
\mathit{tcc}_{\rho}(t\ \ \mathbf{typed}\ \ u) & = & \mathit{tcc}_{\rho}\, t \wedge \mathit{tcc}_{\rho}\, u \wedge \\
 & & \qquad (\mathit{type_pred}_{\rho}\, t \Rightarrow \mathit{inject_type}_{\rho}\, t\ u)
\end{array}
$$

Figure 9.5: Specification of Type Correctness Conditions

Logical Context The type correctness rule for an implication term $p \Rightarrow q$ assumes p when typing checking the term q. Intuitively, this is satisfactory since the value of $p \Rightarrow q$ only depends on the value of q when p is true. The complete set of logical context rules are shown below.

Term	Context for p	Context for q
$p \Rightarrow q$		p
$p \wedge q$		p
$p \vee q$		$\neg p$
cond $b\ p\ q$	b	$\neg b$

The context for $\wedge$, $\vee$ and $\Rightarrow$ can be derived from the rule for **cond** using the equivalences given in Section 9.1.2.

Type Context The type correctness rules for application, equality and annotation include context deduced from the type of one of the terms. Consider the application of a constant $x : \{p\}$ to a function $f : (\Pi\{p\} \cdot T)$: the type correctness condition must show that x is in the domain of f, i.e. that $p\ x$, but this can be deduced from the type of x. The full type correctness condition is therefore $p\ x \Rightarrow p\ x$.

The type context of term t is $type_pred_\rho\, t$ where $type_pred : TypEnv \to \mathcal{T} \to \mathcal{T}$ is specified by:

$$\begin{array}{l} type_pred_\rho\, u = type_pred'_\rho\, u\ (type_inf_\rho\, u) \\ \textbf{where}\ \begin{array}{lcl} type_pred'_\rho\, u\ \{p_A\} & = & p\ u \wedge type_pred'_\rho\, u\ A \\ type_pred'_\rho\, u\ U & = & \mathbf{t} \end{array} \end{array}$$

9.4.6 Injecting a Term into a Type

The function $inject_type : TypEnv \to \mathcal{T} \nrightarrow \mathcal{T} \nrightarrow \mathcal{T}$ gives the hypothesis for injecting a term t into a type u. The hypothesis is $inject_type_\rho\, t\ u$, provided that both types are well-formed: $type_ok_\rho\, t$, $type_ok_\rho\, u$, and that their kinds are appropriate:

$$type_inf_\rho\, t = type_inf_\rho(type_inf_\rho\, u)$$

The function is defined by the equations shown in Figure 9.6. The definition makes use of the auxiliary function **domain_of** defined in Section 9.4.7.

For some forms of U, the hypothesis for injecting $t : T$ into U can be rewritten to a simpler form; a default case of $inject_type_\rho\, t\ U$ applies otherwise, requiring $T = U$. The weaker 'subtype' condition $T \subseteq U$ would be sufficient condition but such a relationship between types cannot be expressed in the logic.

Covariant Type Constructors Figure 9.6 include a case which applies to types of the form $F\ A$ where the type constructor F is *covariant*. If F is a covariant type

$$
\begin{array}{lcl}
\mathit{inject_type}_\rho\, t\ u & = & \textbf{t provided } \mathit{type_inf}_\rho\, t \equiv u \\
\mathit{inject_type}_\rho\, f\ (\Pi t \cdot u) & = & (\textbf{domain_of}(\mathit{type_inf}_\rho\, f) = t) \wedge \\
 & & \quad (\forall t.\, \mathit{type_pred}_{\rho+t}\, \underline{0} \Rightarrow \mathit{inject_type}_{\rho+t}(f^{+}\ \underline{0})\ u) \\
 & & \quad \textbf{provided is_product}(\mathit{type_inf}_\rho\, f) \\
\mathit{inject_type}_\rho\, t\ \{p\} & = & p\ t \wedge \mathit{inject_type}_\rho\, t\ (\textbf{domain_of}\ (\mathit{type_inf}_\rho\, p)) \\
\mathit{inject_type}_\rho\, t\ (\textbf{cond}\, b\ u\ u') & = & (b \Rightarrow \mathit{inject_type}_\rho\, t\ u) \wedge (\neg b \Rightarrow \mathit{inject_type}_\rho\, t\ u') \\
\mathit{inject_type}_\rho\, t\ (u\ \textbf{typed}\ v) & = & \mathit{inject_type}_\rho\, u\ v \Rightarrow \mathit{inject_type}_\rho\, t\ u \\
\mathit{inject_type}_\rho\, t\ (F\ A) & = & (\forall A'.\, \mathit{inject_type}_{\rho+A'}\, \underline{0}\ A^{+}) \\
 & & \quad \textbf{provided } \mathit{type_inf}_\rho\, t = F\ A' \\
 & & \quad \textbf{and } F \text{ is covariant}
\end{array}
$$

Otherwise, when none of the above cases apply:

$$
\mathit{inject_type}_\rho\, t\ u \quad = \quad (\mathit{type_inf}_\rho\, t = u)
$$

Figure 9.6: Specification of Type Injection

constructor and A, A' are types then elements of $F\ A$ are also elements of $F\ A'$ if elements of A are elements of A'. It is conjectured that the well-formation rules on datatypes are sufficient to ensure that all datatypes have this property. The covariance of the pair type constructor (see Section 3.4.2) can be built into the type checker using the rule:

$$
\begin{array}{rl}
\mathit{inject_type}_\rho\, p\ ((PAIR)\ A\ f_A) = & \mathit{inject_type}_\rho(fst\ p)\ A \wedge \\
 & \mathit{inject_type}_\rho(snd\ p)\ (f_A\ (fst\ p))
\end{array}
$$

Not all type constructors in λC are covariant. In particular, the function type constructor Π is not covariant in its first component: for example, elements of $(\Pi\mathbb{N} \cdot \mathbb{N})$ are not elements of $(\Pi\mathbb{Z} \cdot \mathbb{N})$. This occurs since all functions are total.

9.4.7 Auxiliary Functions

The auxiliary functions used in the definitions above are defined below.

Sorts The function **kind_or_type** : $\mathcal{T}_i \rightarrow \mathbb{B}$ satisfies:

$$
\begin{array}{lcl}
\textbf{kind_or_type}\,\Box & = & \textbf{t} \\
\textbf{kind_or_type}\,* & = & \textbf{t} \\
\textbf{kind_or_type}\{\ p_A\ \} & = & \textbf{kind_or_type}\ A
\end{array}
$$

for other terms it is false.

Product Terms The predicate **is_product** : $\mathcal{T}_i \to \mathbb{B}$ holds for product terms:

$$\textbf{is_product}(\Pi T \cdot U) = \mathbf{t}$$

for other terms it is false.

Predicate The predicate **is_pred_type** : $\mathcal{T}_i \to \mathbb{B}$ holds for product terms with range type boolean:

$$\textbf{is_pred_type}(\Pi T \cdot \mathbb{B}) = \mathbf{t}$$

for other terms it is false.

Boolean Functions The predicate **is_boolfun_type** : $\mathcal{T}_i \to \mathbb{B}$ holds for the boolean type or product terms boolean range:

$$\begin{array}{lcl} \textbf{is_boolfun_type}(\Pi T \cdot U) & = & \textbf{is_boolfun_type}\, U \\ \textbf{is_boolfun_type}\, \mathbb{B} & = & \mathbf{t} \end{array}$$

for other terms it is false.

Product Domain The function **domain_of** : $\mathcal{T}_i \nrightarrow \mathcal{T}_i$ returns the domain type of a product term:

$$\textbf{domain_of}(\Pi T \cdot U) = T$$

Product Range The function **range_of** : $\mathcal{T}_i \nrightarrow \mathcal{T}_i \to \mathcal{T}_i$ returns the range type of a product term for a particular value in the domain type:

$$\textbf{range_of}(\Pi T \cdot U)\ t = U[t^+]^-$$

Base Type The function **base_type_of** : $\mathcal{T}_i \to \mathcal{T}_i$ strips constraints from a subtype:

$$\begin{array}{lcl} \textbf{base_type_of}\{p_A\} & = & \textbf{base_type_of}\, A \\ \textbf{base_type_of}\, t & = & t \end{array}$$

9.4.8 Rewrite Rules

This section describes the rewrite rules used by the type checker. The rewrite rules define a reduction function; the type checker uses lazy evaluation of this so that a reduction is only made when necessary. This preferable, since the expansion of definitions in reduction may cause a term to become very large.

The reductions implemented in the typechecker are:

Standard Conversions These are β-reduction and η-reduction:

$$\begin{array}{lcl} (\lambda A \cdot b)\ a & \rightsquigarrow & b[0 := a^+]^- \\ (\lambda A \cdot b\ \underline{0}) & \rightsquigarrow & b^- \ \ \textbf{provided}\ 0 \notin \Phi\, b \end{array}$$

Since terms are represented using indices for variables, α-conversion is not required.

Expansion of Definitions Reduction replaces a defined identifier by its definition, in the type environment $\rho = (\kappa, \nu)$:

$$l.x \rightsquigarrow t \textbf{ where } \kappa\ l.x = Defn\ t\ T$$

Conditional

$$\textbf{cond}\,\textbf{t}\ t\ u \rightsquigarrow t \quad \textbf{cond}\,\textbf{f}\ t\ u \rightsquigarrow u$$

Annotation An annotation can be discarded:

$$u\ \textbf{typed}\ U \rightsquigarrow u$$

Pairs The following reductions for pairs can be added:

$$fst\ ((\textbf{pair}\ A \cdot B)\ t_1\ t_2) \rightsquigarrow t_1$$
$$snd\ ((\textbf{pair}\ A \cdot B)\ t_1\ t_2) \rightsquigarrow t_2$$

Although pairs are not a basic form of term, there is no general technique for making the typechecker aware of properties of declared functions.

9.5 Type Errors

A number of the type assignment rules have decidable premises, which are included in the *type_ok* function. For example, the product rule requires that the type assigned to each component of the product is either $*$ or $\Box$. A type checker can signal an error if this condition and other similar conditions are not satisfied.

However, other type errors, such as the application of the term 1 to a function on booleans, would not be identified in this way. This occurs because this application can be considered to be well-typed by using the conversion rule to convert the type of 1 from $\mathbb{N}$ to the required type $\mathbb{B}$. The type correctness condition $\mathbb{B} = \mathbb{N}$ results. We describe an enhancement to the type checking algorithm which reports this as a type error, avoiding unprovable correctness conditions.

9.5.1 Type Compatibility

We introduce the compatibility relation $\approx$ between types which holds only if two types are the same *when subtypes are ignored.* Inference rules which specify this relation are shown in Figure 9.7, assuming that for $t \approx u$, the type t and u are in the partial normal form. Figure 9.7 includes a case for all terms. This may seem unnecessary, since only 'types' are checked for compatibility and some terms, such as the λ-abstraction cannot denote a type. However, new constants can be introduced which construct types from one or more arguments. To show that types constructed in this way are compatible, the compatibility of their arguments must be shown and any term could be used as an argument.

(Equivalence) $\dfrac{A \equiv B}{A \approx B}$

(Abstraction) $\dfrac{A \approx A' \quad B \approx B'}{(\lambda x : A \cdot B) \approx (\lambda y : A' \cdot B')}$

(Product) $\dfrac{A \approx A' \quad B \approx B'}{\Pi A \cdot B \approx \Pi A' \cdot B'}$

(Application) $\dfrac{f \approx f' \quad a \approx a'}{f\ a \approx f'\ a'}$

(Subtype) $\dfrac{A \approx A'}{\{A \mid p\} \approx \{A' \mid q\}}$

(Implies) $\dfrac{p \approx q \quad p' \approx q'}{(p \Rightarrow q) \approx (p' \Rightarrow q')}$

(Forall) $\dfrac{f \approx g}{\forall f \approx \forall g}$

(Choice) $\dfrac{f \approx g}{\epsilon f \approx \epsilon g}$

(Equals) $\dfrac{A \approx A' \quad B \approx B'}{(A = B) \approx (A' = B')}$

(Conditional) $\dfrac{p \approx p' \quad A \approx A' \quad B \approx B'}{\mathbf{cond}\, p\ A\ B \approx \mathbf{cond}\, P'\ A'\ B'}$

(Annotation) $\dfrac{A \approx A'}{A\ \mathbf{typed}\ B \approx A'\ \mathbf{typed}\ B'}$

Two types are compatible if they are identical, or they have compatible subterms or if they become compatible when subtype constraints are discarded. The 'subtype' rule can be applied when only one type is a subtype by viewing any type as a subtype with the constraint 'true'.

Figure 9.7: Specification of Type Compatibility

$$
\begin{array}{lcl}
type_ok_\rho(f\ a) & = & type_ok_\rho\, f \wedge type_ok_\rho\, a \wedge \\
 & & \quad \textbf{let } F = type_inf_\rho\, f \textbf{ in} \\
 & & \qquad \textbf{is_product } F \wedge \textbf{domain_of } F \approx type_inf_\rho\, a \\
type_ok_\rho(\textbf{cond}\, p\ t\ u) & = & type_ok_\rho\, p \wedge type_inf_\rho\, p \equiv \mathbb{B} \wedge \\
 & & \quad type_ok_\rho\, t \wedge type_ok_\rho\, u \\
 & & \qquad type_inf_\rho\, t \approx type_inf_\rho\, u \\
type_ok_\rho(t = u) & = & type_ok_\rho\, t \wedge type_ok_\rho\, u \wedge \\
 & & \quad type_inf_\rho\, t \approx type_inf_\rho\, u \\
type_ok_\rho(t\ \textbf{typed}\ u) & = & type_ok_\rho\, t \wedge type_ok_\rho\, u \wedge \textbf{kind_or_type}(type_inf_\rho\, u) \wedge \\
 & & \quad type_inf_\rho\, t \not\equiv \Box \wedge \\
 & & \quad type_inf_\rho(type_inf_\rho\, t) \approx type_inf_\rho\, u)
\end{array}
$$

Figure 9.8: Specification of Type Checking with Compatibility

9.5.2 Type Checking with Compatibility

We choose to restrict the use of type injection to cases where the required type and the inferred type are compatible; other cases are treated as type errors. This approach restricts the occurence of unproveable type correctness conditions. Figure 9.8 shows a modified specification of the *type_ok* function following this approach. Cases not shown are the same as those given in the original specification, Figure 9.3.

9.5.3 A Summary of Type Errors

The type checker returns an error message whenever $type_ok_\rho\, t$ is false. Each possible type error corresponds to one of the conjuncts being false, excluding those conjuncts which are required to check that each subterm has a type. For each term, Figure 9.9 shows the conjuncts of $type_ok_\rho$ and describes the resulting type error.

9.6 Inference of Implicit Arguments

Implicit arguments of polymorphic functions are inferred by a simple form of unification during type checking. In this section, an outline of the elaboration of the type checking algorithm to infer implicit arguments is given.

Term	*type_ok* conjunct	Error
$f\ a$	**is_product** ρF	f is not a function
$f\ a$	**domain_of** $F \approx type_inf_{\rho}\ a$	the type of a is not compatible with the domain type of f
$\lambda t \cdot u$	**kind_or_type**$(type_inf_{\rho}\ t)$	t is not a type or kind
$\Pi t \cdot u$	**kind_or_type**$(type_inf_{\rho}\ t)$	t is not a type or kind
$\Pi t \cdot u$	**kind_or_type**$(type_inf_{\rho+t}\ u)$	u is not a type or kind
$\{t\}$	**is_pred_type**$(type_inf_{\rho}\ t)$	t is not a predicate
$p \Rightarrow q$	$type_inf_{\rho}\ p \equiv \mathbb{B}$	p is not a boolean term
$p \Rightarrow q$	$type_inf_{\rho}\ q \equiv \mathbb{B}$	q is not a boolean term
$\forall f$	**is_pred_type**$(type_inf_{\rho}\ f)$	f is not a predicate
ϵf	**is_pred_type**$(type_inf_{\rho}\ f)$	f is not a predicate
cond $p\ t\ u$	$type_inf_{\rho}\ p \equiv \mathbb{B}$	p is not a boolean term
cond $p\ t\ u$	$type_inf_{\rho}\ t \approx type_inf_{\rho}\ u$	the types of t and u are not compatible
$t = u$	$type_inf_{\rho}\ t \approx type_inf_{\rho}\ u$	the types of t and u are not compatible
t **typed** u	**kind_or_type**$(type_inf_{\rho}\ u)$	u is not a kind or type
t **typed** u	$type_inf_{\rho}(type_inf_{\rho}\ t) \equiv type_inf_{\rho}\ u$	the type of t and u are of different kinds

Figure 9.9: Type Errors

9.6.1 Instantiation Variables

An instantiation variable is added to the $\mathcal{T}$ datatype. Each variable $?n$ is tagged with its type, which may contain other instantiation variables.

$$\mathcal{T}_i ::= \ldots \mid ?\mathbb{N} : \mathcal{T}_i$$

The first step is to add instantiation variables to terms as place holder for the implicit arguments of constants. The type environment is elaborated to associate each constant with the number of implicit arguments:

$$\mu \in NumDefs == \mathcal{I} \times \mathcal{I} \rightarrow \mathbb{N}$$

The function:

$$add_ivars : ConTyp \times NumDefs \rightarrow \mathcal{T}_i \rightarrow (\mathcal{T}_i \times \mathbb{N})$$

specifies how instantiation variables are added. An instantiation variable replaces each implicit argument: each instantiation variable is different.

$$\begin{array}{lcl}
add_ivars_{\kappa,\mu}\ i\ * & = & (*, i) \\
add_ivars_{\kappa,\mu}\ i\ \underline{j} & = & (\underline{j}, i) \\
add_ivars_{\kappa,\mu}\ i\ l.c & = & con_ivars(\mu\ l.c)\ (\kappa\ l.c)\ i\ l.c \\
add_ivars_{\kappa,\mu}\ i\ (f\ a) & = & (f'\ a', k) \\
 & & \textbf{where}\ add_ivars_{\kappa,\mu}\ \ i\ f = (f', j) \\
 & & \qquad\quad add_ivars_{\kappa,\mu}\ \ j\ a = (a', k) \\
add_ivars_{\kappa,\mu}\ i\ (\lambda t \cdot u) & = & ((\lambda t' \cdot u'), k) \\
 & & \textbf{where}\ add_ivars_{\kappa,\mu}\ \ i\ t = (t', j) \\
 & & \qquad\quad add_ivars_{\kappa,\mu}\ \ j\ u = (u', k)
\end{array}$$

Only constants have implicit arguments. The function con_ivars add instantiation variables to a constant. In $con_ivars\ \ n\ \ t\ \ i\ \ f$, n is the number of implicit arguments, t is a term representing the types of the implicit arguments, i is the index of the next instantiation variable and f is the term to which the implicit arguments are applied.

$$\begin{array}{lcl}
con_ivars\, 0\ \ t\ \ i\ \ t' & = & (t', i) \\
con_ivars(n+1)\ (\Pi t \cdot u)\ i\ f & = & con_ivars\, n\ \ (u[0 := ?i : t^{+}])^{-}\ \ (i+1)\ \ (f\ \ (?i : t))
\end{array}$$

The specification above does not take account of the way implicit arguments can be made explicit. This feature is straight forward.

Example The function fst is declared as:

$$fst\ [2] : A : * \rightarrow f : (a \rightarrow *) \rightarrow PAIR\ (\lambda a : A \cdot f\ a) \rightarrow A$$

When instantiation variables replace the implicit arguments in the term $fst\ p$, the result is:

$$fst\ (?1 : *)\ (?2 : ((?1 : *) \rightarrow *))\ p$$

9.6.2 Type Checking with Unification

Unification is used during type checking to instantiate the instantiation variables. The predicate $type_oki_{\rho}^{b,t'}\ t$ indicates whether the term t is well-typed, where:

b	is a binding $b \in \mathcal{B} == \mathbb{N} \nrightarrow \mathcal{T}_i$
t'	is the term t with variables instantiated

We write $t[b]$ for the substitution of the binding b in the term t. The most important cases of this predicate are specified in Figure 9.10.

In the predicate $type_oki_{\rho}^{b,t'}\ t$, some of the instantiation variables which occur in t may be instantiated, giving the binding b. These instantiation variables are removed by

$$
\begin{array}{lcl}
\mathit{type_oki}_{\rho}^{\{\},*}\ * & = & \mathbf{t} \\
\mathit{type_oki}_{\rho}^{\{\},\underline{i}}\ \underline{i} & = & \mathbf{t} \\
\mathit{type_oki}_{\rho}^{\{\},(?i:t)}\ ?i : t & = & \mathbf{t} \\
\mathit{type_oki}_{\rho}^{\{\},l.c}\ l.c & = & \mathbf{t} \\
\mathit{type_oki}_{\rho}^{b,t}(f\ a) & = & \mathit{type_oki}_{\rho}^{b_1,f'}\ f \wedge \mathbf{is_product}(\mathit{type_inf}_{\rho}\, f') \wedge \\
 & & \quad \mathit{type_oki}_{\rho}^{b_2,a'}\ a[b_1] \wedge \\
 & & \quad \mathbf{domain_of}(\mathit{type_inf}_{\rho}\, f'[b_2])\ \mathcal{U}_{\rho}^{b_3}\ \mathit{type_inf}_{\rho}\, a' \\
 & & \mathbf{where}\ b = b_1 \cup b_2 \cup b_3 \\
 & & \qquad\ \ t = f'[b_2 \cup b_3]\ a'[b_3] \\
\mathit{type_oki}_{\rho}^{b,v}(\lambda t \cdot u) & = & \mathit{type_oki}_{\rho}^{b_1,t'}\ t \wedge \mathbf{kind_or_type}(\mathit{type_inf}_{\rho}\, t') \wedge \\
 & & \quad \mathit{type_oki}_{\rho+t'}^{b_2,u'}\ u[b_1^+] \\
 & & \mathbf{where}\ b = b_1 \cup b_2^- \\
 & & \qquad\ \ v = (\lambda t'[b_2^-] \cdot u')
\end{array}
$$

Figure 9.10: Type Checking with Unification

substitution from t, yielding t'. It might appear simpler to return only the binding b, replacing occurences of t' by $t[b]$. The reason this simplification is not satisfactory is the scope adjustments required to type check terms such as $(\lambda t \cdot u)$. The binding generated by checking u must be decremented: all the terms in the binding are decremented *and any variable mapped to a term containing the variable* $\underline{0}$ *is discarded.*

9.6.3 Unification of Types

In the specification of *type_oki*, the type compatibility relation $\approx$ (specified in Figure 9.7) is replaced by a restricted form of unification between type. Types match exactly as specified in Figure 9.7, instantiating instantiation variables where possible. The most important cases of the unification specification are shown below:

$$
\begin{array}{lcl}
?n : t\ \mathcal{U}_{\rho}^{\{n \mapsto u\} \cup b}\ u & = & t\ \mathcal{U}_{\rho}^{b}\ (\mathit{type_inf}_{\rho}\, u) \\
 & & \mathbf{provided}\ ?n \text{ does not occur in } u \\
u\ \mathcal{U}_{\rho}^{\{n \mapsto u\} \cup b}\ ?n : t & = & t\ \mathcal{U}_{\rho}^{b}\ (\mathit{type_inf}_{\rho}\, u) \\
 & & \mathbf{provided}\ ?n \text{ does not occur in } u \\
(\Pi t \cdot u)\ \mathcal{U}_{\rho}^{b_1 \cup b_2^-}\ (\Pi t' \cdot u') & = & t\ \mathcal{U}_{\rho}^{b_1}\ t' \wedge u[b_1^+]\ \mathcal{U}_{\rho+t[b_1]}^{b_2}\ u'[b_1^+] \\
(f\ a)\ \mathcal{U}_{\rho}^{b_1 \cup b_2}\ (f'\ a') & = & f\ \mathcal{U}_{\rho}^{b_1}\ f' \wedge a[b_1]\ \mathcal{U}_{\rho}^{b_2}\ a'[b_2]
\end{array}
$$

Note that an instantiation variable can only be instantiated to a term of a matching type. If the type does not match, a type error has occurred.

Chapter 10

Datatypes and Recursion

Recursive datatypes and associated terms are described in this chapter. Section 10.1 describes the datatype declaration. A datatype is not a new form of term, instead a datatype declaration stands for a number of constant declarations which model the properties of the datatype. Two new forms of term that can only be used with datatypes are also described: these are the case term, described in Section 10.2, and the primitive recursion term, described in Section 10.3. The final section describes the general recursion term, which allows a function to be constructed recursively, using a measure function to show that the recursion terminates.

10.1 Datatype Declarations

10.1.1 Abstract Syntax

The abstract syntax of a datatype declaration is:

$$\mathcal{D}_x ::= \ldots \mid \mathcal{I}\ \mathbf{seq}(\mathcal{I} : \mathcal{T}_x)\ ::=\ \mathbf{seq}\,\mathcal{C}_x$$

$$\mathcal{C}_x ::= \mathcal{I} \mid (\mathcal{I},\ \mathbf{seq}(\mathcal{I} : \mathcal{T}_x),\ \mathcal{I})$$

Example The list datatype declaration is represented in the abstract syntax as:

$$\mathit{List}\ \langle(A : *)\rangle ::= \langle(\mathit{null}, \langle\rangle, \mathit{empty}), (\mathit{cons}, \langle \mathit{head} : A, \mathit{tail} : \mathit{List}\ A\rangle, \mathit{nonempty})\rangle$$

Identifiers A datatype declaration:

$$\begin{aligned} t\ \langle p_1 : P_1, \ldots, p_I : P_I\ \rangle ::= \langle & (c_1, \langle a_{11} : A_{11}, \ldots, a_{1K} : A_{1K}\rangle,\ r_1), \\ & \vdots, \\ & (c_J, \langle a_{J1} : A_{J1}, \ldots, a_{JK} : A_{JK}\rangle,\ r_J)\rangle \end{aligned}$$

introduces the following identifiers:

(i) the type itself t,

(ii) a list of parameters p_i for $i \in 1 \ldots I$,

(iii) a *constructor* c_j for each case $j \in 1 \ldots J$,

(iv) an *accessor* a_{jk}, or de-constructor, for each component of each case $k \in 1 \ldots K_j$,

(v) a *recogniser* predicate r_j for each case.

For cases with no accessors, the recogniser is optional.

Dependency The types used in the data type declaration can be dependent.

1. The datatype parameter p_i can be free in either the type of a subsequent parameter P_j *where* $i < j$ or in an accessor type A_{jk}.
2. The accessor identifier a_{jk} can be free in the type A_{jl} (*where* $k < l$) of a subsequent accessor of the same case.

The first form of dependency occurs in the list datatype, where the parameter is the element type. The second form allows the cases of the datatype to behave like record types, where the type of one of the field depends on the value of an earlier one. This feature is not used in the list datatype.

10.1.2 Visibility in Datatype Declarations

A datatype declaration is equivalent to a number of simple declarations. The function $adt : \mathcal{D}_x \to \mathcal{D}_x$ describes the datatype expansion. The function adt is specified by:

$$adt(t\ ps ::= \langle c_1, \ldots, c_J \rangle) = t : (\Pi \mathbin{/} ps \bullet *); adt_case\ t\ ps\ c_1; \ldots adt_case\ t\ ps\ c_J$$

If ps is a sequence of declarations and A is a term, the iterated product is written $\Pi \mathbin{/} ps \bullet A$. For example:

$$\Pi \mathbin{/} \langle p_1 : P_1, p_2 : P_2 \rangle \bullet A = (\Pi p_1 : P_1 \cdot (\Pi p_2 : P_2 \cdot A))$$

The function *adt_case* specifies the expansion of each case:

$$
\begin{array}{lcl}
adt_case\ t\ ps\ (c) & = & c : (\Pi \ / \ ps \bullet t) \\
adt_case\ t\ ps\ (c, as,\ r) & = &
\end{array}
$$

$$
\begin{array}{l}
r : (\Pi \ / \ ps \bullet (\Pi t' : t\ p_1 \ldots p_I \cdot \mathbb{B})); \\
c : (\Pi \ / \ ps \frown as \bullet \{r\ p_1 \ldots p_I\}); \\
a_1 : (\Pi \ / \ ps \bullet (\Pi c : \{r\ p_1 \ldots p_I\} \cdot A_1)); \\
a_2 : (\Pi \ / \ ps \bullet (\Pi c : \{r\ p_1 \ldots p_I\} \cdot A_2[a_1 := a_1\ p_1 \ldots p_I\ c])); \\
\vdots \\
a_K : (\Pi \ / \ ps \bullet (\Pi t' : \{r\ p_1 \ldots p_I\} \cdot \\
\quad A_K[a_1 := a_1\ p_1 \ldots p_I\ t'] \ldots [a_{K-1} := a_{K-1}\ p_1 \ldots p_I\ t'])) \\
\textbf{where}\ \ ps = \langle p_1 : P_1, \ldots, p_I : P_I \rangle \\
\qquad\qquad as = \langle a_1 : A_1, \ldots, a_K : A_K \rangle
\end{array}
$$

The order of the declarations is significant: the recogniser function r is first, followed by the constructor c and the accessors a_k. The recogniser predicate is used in the type of the constructor function, so that the type shows that a value constructed by c_j belongs to the cth case. The accessor functions can only be applied to a datatype value of the correct case: the recogniser predicate is used to specify the domain of the function.

10.1.3 Visibility Rules Datatype Declarations

All the visibility rules described in Chapter 8 apply to the declarations generated by expanding a datatype declaration. This ensures that the identifiers for the datatype t, constructors c_j, recognisers r_j and accessors a_{jk} are distinct and have no existing declaration:

$$
[\![t\ ps ::= cs]\!]^l_\rho = [\![adt(t\ ps ::= cs)]\!]^l_\rho\ \textbf{provided}\ \ adt_ok(t\ ps ::= cs)
$$

To ensure that the declarations generated by the expansion do not interfere, there are four additional rules[1]:

1. The datatype parameter identifiers p_i do not overlap with t, c_j, r_j or a_{jk}; together with the standard visibility rules, this is sufficient to ensure that none of the identifiers t, c_j, r_j or a_{jk} appears free in the parameter types P_i.

2. The constructor c_j and recogniser r_j of each case may not be free in subsequent terms generated by the expansion, nor may an accessor for one case be free in a subsequent case.

3. There must be at least one case – a *base* case – of the datatype in which the datatype t is not free in any accessor type.

[1] It is conjectured that these rules are sufficient to ensure that all datatypes are 'covariant', as described in Section 9.4.6.

$$
\begin{aligned}
adt_ok(t\ ps ::= cs) \;&=\; adt_case_ok_{\emptyset,t,ps}\, cs \wedge (\exists c \in cs \cdot adt_base_case_t\, cs) \wedge \\
&\qquad t \notin \{\ p_1, \ldots, p_I\ \} \ \textbf{where}\ ps = \langle p_1 : P_1, \ldots, p_1 : P_1 \rangle
\end{aligned}
$$

$$
\begin{aligned}
adt_base_case_t(c) \;&=\; \mathbf{t} \\
adt_base_case_t(c, \langle a_1 : A_1, \ldots, a_K : A_K \rangle, r) \;&=\; \textstyle\bigwedge_{k \in 1 \ldots K} t \notin \Phi_{\mathcal{I}}\ A_k
\end{aligned}
$$

$$
\begin{aligned}
&adt_case_ok_{\eta,t,ps}\langle\rangle \;=\; \mathbf{t} \\
&adt_case_ok_{\eta,t,ps}\langle c \rangle \frown cs \;=\; c \notin \{p_1, \ldots, p_I\} \wedge adt_case_ok_{\eta',t,ps}\, cs \\
&\quad \textbf{where}\ ps = \langle p_1 : P_1, \ldots, p_I : P_I \rangle,\ \eta' = \eta \cup \{c\} \\
&adt_case_ok_{\eta,t,ps}\langle (c\ as\ r) \rangle \frown cs \;= \\
&\quad \{c, r\} \cap \{p_1, \ldots, p_I\} = \emptyset \wedge adt_accessors_ok_{\eta',t,ps}\, as \wedge adt_case_ok_{\eta'',t,ps}\, cs \\
&\quad \textbf{where}\ \ as = \langle a_1 : A_1, \ldots a_K : A_K \rangle,\ ps = \langle p_1 : P_1, \ldots, p_I : P_I \rangle \\
&\qquad\qquad \eta' = \eta \cup \{r, c\},\ \eta'' = \eta' \cup \{a_1, \ldots, a_K\} \\
&adt_accessors_ok_{\eta,t,ps}\langle\rangle \;=\; \mathbf{t} \\
&adt_accessors_ok_{\eta,t,ps}\langle a : A \rangle \frown as \;=\; \Phi_{\mathcal{I}}\, A \cap \eta = \emptyset \wedge \\
&\quad (t \in \Phi_{\mathcal{I}}\, A \Rightarrow A = t\ p_1 \ldots p_I \wedge adt_accessors_ok_{\eta',t,ps}\, as) \wedge \\
&\quad (t \notin \Phi_{\mathcal{I}}\, A \Rightarrow adt_accessors_ok_{\eta,t,ps}\, as) \\
&\quad \textbf{where}\ ps = \langle p_1 : P_1, \ldots, p_I : P_I \rangle,\ \eta' = \eta \cup \{a\}
\end{aligned}
$$

Notes 1. The function $\Phi_{\mathcal{I}} : \mathcal{T}_x \rightarrow \mathbb{P}\,\mathcal{I}$ returns the set of directly visible free identifiers of a term. It is similar to the function Φ defined on page 121.

Figure 10.1: Additional Datatype Visibility Rules

4. The datatype t may be free in the term specifying the type of an accessor, provided that:
   (a) t is parameterised with the formal parameters $p_1 \ldots p_I$,
   (b) no subsequent accessor has a type which depends on this accessor.

If the datatype t is free in the type of an accessor, the case is termed 'recursive'. The recursive occurence of the type t cannot be an argument to a type constructor[2]. The predicate *adt_ok*, shown in Figure 10.1, specifies these additional checks. In the predicate $adt_case_ok_{\eta,t,ps}$, which specifies the checks for one case of the datatype, the set η contains the identifiers which have restricted usage; t is the datatype identifier and *ps* the list of parameters. The predicate $adt_accessors_ok_{\eta,t,ps}$ checks an accessor list.

[2]This restriction is sufficient to ensure that the recursive type exists. PVS allows certain type constructors such as lists to be used; in general, any 'positive' uses of the type could be allowed [Sha93].

10.1.4 Auxiliary Functions for Datatypes

A number of auxiliary functions associated with datatypes which are needed below, are introduced here.

1. The constant $l.d$ is a datatype if $is_adt_\rho(l.d)$.

$$is_adt : TypEnv \to (\mathcal{I} \times \mathcal{I}) \to \mathbb{B}$$

2. The set of constructors for the datatype $l.d$ is given by $adt_cons_ids_\rho(l.d)$.

$$adt_cons_ids : TypEnv \twoheadrightarrow (\mathcal{I} \times \mathcal{I}) \to \mathbb{P}(\mathcal{I} \times \mathcal{I})$$

3. The type of a datatype constructor $l.c$ when the datatype parameters types are ps is given by $adt_cons_type_\rho(l.c)\ ps$.

$$adt_cons_type : TypEnv \twoheadrightarrow (\mathcal{I} \times \mathcal{I}) \twoheadrightarrow \mathbf{seq}\,\mathcal{T}_i \to \mathcal{T}_i$$

4. The number of accessors for the datatype case with constructor $l.c$ and parameter types ps is given by $adt_num_accs_\rho\ c\ ps$.

$$adt_num_accs : TypEnv \twoheadrightarrow (\mathcal{I} \times \mathcal{I}) \twoheadrightarrow \mathbf{seq}\,\mathcal{T}_i \to \mathbb{N}$$

5. The recogniser function for the case of a datatype with constructor $l.c$ is given by $adt_recogniser_\rho\ l.c$.

$$adt_recogniser : TypEnv \twoheadrightarrow (\mathcal{I} \times \mathcal{I}) \twoheadrightarrow (\mathcal{I} \times \mathcal{I})$$

6. The set of accessors for the case of the datatype $l.d$ with constructor $l.c$ is given by $adt_acc_ids_\rho\ l.d\ l.c$.

$$adt_acc_ids : TypEnv \twoheadrightarrow (\mathcal{I} \times \mathcal{I}) \twoheadrightarrow (\mathcal{I} \times \mathcal{I}) \twoheadrightarrow \mathbb{P}(\mathcal{I} \times \mathcal{I})$$

It is not possible to specify all of these functions fully since the type environment described in Chapter 8 does not distinguish datatypes from other types. In particular, the association between the datatype and the constructor, recogniser and accessor functions used to model the datatype is not preserved in the environment. We do not present the necessary elaboration of the environment here.

10.2 Case Terms

A datatype value can be decomposed into its different cases using a case term. A case term may also be applied to a term which belongs to a subtype of a datatype, defined by excluding some of the datatype constructors. A type correctness condition may be produced to show that the term does not belong to one of the omitted cases. Case terms are convenient but not necessary, since the recogniser and accessor functions of the datatype can be combined to have the same effect as a case term.

10.2.1 Abstract Syntax

The abstract syntax with variables for the case term is:

$$\mathcal{T}_x ::= \ldots \mid \textbf{case} \;\; \mathcal{T}_x \;\; \underset{j}{[\!]} \;\; \mathcal{I}\;\mathcal{I}\ldots\mathcal{I} \Longrightarrow \mathcal{T}_x$$

In the case term ($\textbf{case}\ t\,[\!]_j\, c_j\ a_{j1}\ldots a_{jK} \Longrightarrow u_j$) we refer to the term t as the *case expression*. For each j, there is a *case choice* $c_j\ a_{j1}\ldots a_{jK} \Longrightarrow u_j$, with *case pattern* $c_j\ a_{j1}\ldots a_{jK}$, consisting of a *case identifiers* c_j and *accessor variables* a_{jk}. The term u_j is the *choice term*.

10.2.2 Visibility of Identifiers

The abstract syntax with depth indices is:

$$\mathcal{T}_i ::= \ldots \mid \textbf{case} \;\; \mathcal{T}_i \;\; \underset{j}{[\!]} \;\; (\mathcal{I}.\mathcal{I})\ \mathbb{N} \Longrightarrow \mathcal{T}_i$$

In this syntax the accessor variables are not required and the case identifier is prefixed with the theory identifier. However, it is necessary for the syntax to include the number of accessor variables.

Free Variables The free variables of the case term are specified by:

$$\begin{array}{l}\Phi(\textbf{case}\ t\,[\!]_j\, l.c_j\ K_j \Longrightarrow u_j) = \\ \qquad \Phi\, t \cup \bigcup_{j=1}^{J}\{\ i - K_j \mid i \in \Phi\, u_j \wedge i \geq K_j\ \}\end{array}$$

Encoding The accessor variables of each case choice are bound variables in the choice term[3].

$$\begin{array}{l}[\![\textbf{case}\ t\,[\!]_j\, c_j\ a_{j1}\ldots a_{jK} \Longrightarrow u_j]\!]_\rho = \textbf{case}\ [\![t]\!]_\rho\,[\!]_j[\![c_j]\!]_\rho\ K_j \Longrightarrow [\![u_j]\!]_{\rho_j} \\ \qquad \textbf{where}\ \rho_j \;\; = \;\; \rho + a_{j1} + \ldots + a_{jK}\end{array}$$

10.2.3 Type Correctness

The case term $\textbf{case}\ t\,[\!]_j\, c_j\ a_{j1}\ldots a_{jK} \Longrightarrow u_j$ must satisfy the following type correctness rules:

(i) The case expression t must be a term having as its type a fully parameterised datatype.

[3]This rule requires the case identifier c_j to be visible. Since the syntax does not allow a theory prefix, this requires the datatype to be in a directly visible theory. This constraint could be relaxed.

(ii) The case identifiers c_j must be distinct constructors of the datatype. It is not required that there is a case identifier for all the constructors of the datatype.

(iii) The accessor variables a_{jk} are bound variables which must correspond to the datatype accessors for the case c_j.

(iv) All the choice terms must have the same type.

(v) The accessor variables of the *first* case choice must not be free in the type of the choice term.

In the case term, the type of choice terms is not specified as it is in the primitive recursive term. Since the choice terms may have different numbers of bound variables, the types inferred for the different choices cannot be readily compared. In particular, the accessor variables of the case choice can be free in the type of the choice term. This difficulty is resolved by taking the type of the term on the right hand side of the *first* choice as the type of the terms in all the other choices, with the requirement that the accessor variables are not free in the type of the choice term of the first choice.

10.2.3.1 Type Checking

The *type_ok* function for the case term is specified by:

$$
\begin{aligned}
&\mathit{type_ok}_\rho(\textbf{case}\ t\,[\!]_J\, l.c_j\ K_j \Longrightarrow u_j) = \\
&\quad \mathit{type_ok}_\rho\, t \wedge \mathit{type_inf}_\rho\, t \leadsto d\ p_1 \ldots p_I \wedge \mathit{is_adt}_\rho\, d \wedge \\
&\quad \textstyle\bigwedge_j(\mathit{type_ok}_{\rho_j}\, u_j \wedge K_j = k_j) \wedge \bigwedge_{j=2}^{J}(\mathit{type_inf}_{\rho_j}\, u_j \approx t_j) \wedge \\
&\quad \Phi(\mathit{type_inf}_{\rho_1}\, u_1) \cap \{\ 0 \ldots K_1 - 1\ \} = \emptyset \wedge \\
&\quad (\forall j, j' \in 1 \ldots J \cdot j \neq j' \Rightarrow l.c_j \neq l.c_{j'}) \wedge \\
&\quad \{\ l.c_j | j \in 1 \ldots J\ \} \subseteq \mathit{adt_cons_ids}_\rho\, d \\
&\textbf{where} \\
&\quad l.c_j\ p_1 \ldots p_I \quad \text{has type} \quad (\Pi A_{j1} \cdot \ldots \cdot (\Pi A_{jk_j} \cdot \{\ l.r_j\ p_1 \ldots p_I\ \})) \\
&\qquad\quad \text{where } r_j \text{ is the recogniser for the } j^{th} \text{ case in datatype } d \\
&\quad \rho_j \quad = \quad \rho + A_{j1} + \ldots + A_{jk_j} \\
&\quad t_j \quad = \quad (\mathit{type_inf}_{\rho_1}\ u_1)^{-K_1+K_j}
\end{aligned}
$$

10.2.3.2 Type Inference

The type of the case term is determined by the type of the choice term of the first choice. The *type_inf* function is specified by:

$$
\begin{aligned}
&\mathit{type_inf}_\rho(\textbf{case}\ t\,[\!]_J\, l.c_j\ K_j \Longrightarrow u_j) = (\mathit{type_inf}_{\rho_1}\, u_1)^{-K_1} \\
&\textbf{where} \\
&\quad \mathit{type_inf}_\rho\, t \quad \leadsto \quad d\ p_1 \ldots p_I \\
&\quad l.c_1\ p_1 \ldots p_I \quad \text{has type} \quad (\Pi A_1 \cdot \ldots \cdot (\Pi A_{K_1} \cdot d\ p_1 \ldots p_I)) \\
&\quad \rho_1 \quad = \quad \rho + A_1 + \ldots + A_{K_1}
\end{aligned}
$$

10.2.3.3 Type Correctness Conditions

Type correctness conditions may arise for the case expression and for each choice of the case. Type injection is used to check that all the choice terms have the same type as the choice term of the first case. Universal quantification is required over the case variables, which may be free in the type correctness condition of each choice term. The fact that the case expression belongs to a particular case strengthens the context. In addition, a type correctness condition is required to show that the case expression does not belong to the cases of the datatype for which there is no choice, if any.

$$
\begin{array}{l}
tcc_\rho(\textbf{case } t \,[\!]_J\, l.c_j \; K_j \Longrightarrow u_j) = tcc_\rho\, t \wedge \\
\quad (\forall A_{11} \cdot \ldots \cdot (\forall A_{1K_1} \cdot t^{+K_1} = l.c_1\; p_1 \ldots p_I\; \underline{K_1 - 1} \ldots \underline{0} \Rightarrow tcc_{\rho_1}\, u_1)) \wedge \\
\quad \bigwedge_{j=2}^{J} (\forall A_{j1} \cdot \ldots \cdot (\forall A_{jK_j} \cdot t^{+K_j} = l.c_j\; p_1 \ldots p_I\; \underline{k_j - 1} \ldots \underline{0} \Rightarrow \\
\qquad\qquad tcc_{\rho_j}\, u_j \wedge inject_type_{\rho_j}\, u_j\; v^{-K_1+K_j})) \wedge \\
\quad \bigwedge_{m \in M} (\neg\, is_case_\rho\, p_1 \ldots p_I\; t\; m)
\end{array}
$$

where

$$
\begin{array}{lcl}
type_inf_\rho\, t & \rightsquigarrow & d\; p_1 \;\ldots\; p_I \\
l.c_j\; p_1 \ldots p_I & \text{has type} & (\Pi A_{j1} \cdot \ldots \cdot (\Pi A_{jk_j} \cdot \{\; l.r_j\; p_1 \ldots p_I \;\})) \\
\multicolumn{3}{l}{\quad \text{where } r_j \text{ is the recogniser for the } j^{th} \text{ case in datatype } d} \\
\rho_j & = & \rho + A_{j1} + \ldots + A_{jk_j} \\
v & = & type_inf_{\rho_1}\, u_1 \\
M & = & adt_cons_ids_\rho\, d \setminus \{\; l.c_j | j \in 1 \ldots J \;\}
\end{array}
$$

The function $is_case_\rho\; p_1 \ldots p_I\; c\; t$ forms the predicate that the term t is in case c, with parameters p_i:

$$
is_case_\rho\; p_1 \ldots p_I\; c\; t = \begin{array}{ll}
adt_recogniser_\rho\, c\; p_1 \ldots p_I\; t & \textbf{if } \text{there is a recogniser} \\
t = c\; p_1 \ldots p_I & \textbf{otherwise}
\end{array}
$$

10.2.3.4 Injecting a Term into a Type

A case term can be used to form a type. The type is injected into each case, under an appropriate hypothesis.

$$
\begin{array}{l}
inject_type_\rho\, t\; (\textbf{case } e \,[\!]_J\, l.c_j \; K_j \Longrightarrow u_j) = \\
\quad \bigwedge_J (\forall A_{j1} \cdot \ldots \cdot (\forall A_{jK_j} \cdot e^{+K_j} = l.c_j\; p_1 \ldots p_I\; \underline{k_j - 1} \ldots \underline{0} \Rightarrow inject_type_{\rho_j}\, t^{+K_j}\; u_j))
\end{array}
$$

where

$$
\begin{array}{lcl}
type_inf_\rho\, e & \rightsquigarrow & d\; p_1 \;\ldots\; p_I \\
l.c_j\; p_1 \ldots p_I & \text{has type} & (\Pi A_{j1} \cdot \ldots \cdot (\Pi A_{jk_j} \cdot d\; p_1 \ldots p_I)) \\
\rho_j & = & \rho + A_{j1} + \ldots + A_{jk_j}
\end{array}
$$

10.2.3.5 Reduction

A case term can be reduced when the case expression is an application to an accessor function for which there is a corresponding case choice. Let n be in the range $1 \ldots J$:

$$(\textbf{case}\ l.c_n\ p_1 \ldots p_I\ t_1 \ldots t_{K_n} \mathop{[\!]}_{J} c_j\ K_j \Longrightarrow u_j) \rightsquigarrow$$
$$(u_n[K_n - 1 := t_1^{+K_n}] \ldots [K_n - k := t_k^{+K_n}] \ldots [0 := t_{K_n}^{+K_n}])^{-K_n}$$

10.2.4 Type Errors

The type errors for a case term are summarised in the following table, showing the conjunct of $type_ok_\rho$ which is falsified for each error:

$type_ok$ conjunct	Error
$type_inf_\rho\, t \rightsquigarrow d\ p_1 \ldots p_I \wedge is_adt_\rho\, d$	t is not a term from a datatype
$(\forall j, j' \in 1 \ldots J. j \neq j' \Rightarrow l.c_j \neq l.c_{j'})$	repeated constructor
$\{\ l.c_j \mid j \in 1 \ldots J\ \} \subseteq adt_cons_ids_\rho\, d$	invalid constructor
$K_j = k_j$	wrong number of accessor variables for case j
$\Phi(type_inf_{\rho_1}\, u_1) \cap \{\ 0 \ldots k_1 - 1\ \} = \emptyset$	accessor variable free in the type of the first choice term

Type Compatibility A case for the type compatibility relation $\approx$ is required for the **case** term. The specification is:

$$\text{(Case)}\quad \frac{u_j \approx u'_j \quad t \approx t'}{(\textbf{case}\ t \mathop{[\!]}_J l.c_j\ K_j \Longrightarrow u_j) \approx (\textbf{case}\ t' \mathop{[\!]}_J l.c_j\ K_j \Longrightarrow u'_j)}$$

In addition, the subset of case identifier c_j included in the two terms must be the same.

10.3 Primitive Recursion

The primitive recursion term allows a function on a recursive datatype[4], to be constructed by primitive recursion.

[4] The primitive recursive term can of course be used for non-recursive datatype but it is not necessary in such cases.

10.3.1 Abstract Syntax

The abstract syntax with variables for the primitive recursive term is:

$$\mathcal{T}_x ::= \ldots \mid \textbf{prec}\ \mathcal{I} : \mathcal{T}_x \rightarrow \mathcal{T}_x \mathop{[\!]}_j \mathcal{I}\ \mathcal{I} \ldots \mathcal{I} \Longrightarrow \mathcal{T}_x$$

In the term:

$$\textbf{prec}\ d : t_1 \rightarrow t_2 \mathop{[\!]}_{j \in 1 \ldots J} c_j\ a_{j1} \ldots a_{jK} \Longrightarrow u_j$$

the identifier d is the variable standing for an element of a datatype, t_1 is the domain type of the term and t_2 is the range type, so that the (dependent) function type $\Pi d : t_1.t_2$ is the type of the term. The term has J *cases*, where J is a non-zero integer. In each case, c_j is the *case identifier* and $a_{j1} \ldots a_{jK}$ are the *case variables*: we write K_j for the number of case variables in the jth case. The term u_j is the *case term*.

10.3.2 Visibility of Identifiers

The abstract syntax with depth indices is:

$$\mathcal{T}_i ::= \ldots \mid \textbf{prec}\ \mathcal{T}_i \rightarrow \mathcal{T}_i \mathop{[\!]}_j (\mathcal{I}.\mathcal{I})\ \mathbb{N} \Longrightarrow \mathcal{T}_i$$

In this syntax the variables are not required and the case identifier is prefixed with the theory identifier. However, it is necessary for the syntax to include the number of case variables.

Free Variable The free variables of the primitive recursive term are specified by:

$$\begin{aligned}\Phi(\textbf{prec}\ t_1 \rightarrow t_2 [\!]_j (l.c_j)\ K_j \Longrightarrow u_j) \quad = \quad & \Phi\, t_1 \cup \{\ i-1 \mid i \in \Phi\, t_2 \wedge i > 0\ \} \cup \\ & \textstyle\bigcup_{j=1}^{J} (\{\ i - K_j \mid i \in \Phi\, u_j \wedge i \geq K_j\ \})\end{aligned}$$

Encoding The datatype variable d and the case variables $a_1 \ldots a_{jK}$ in each case are bound variables in the primitive recursive term. The bound variable d is free in the range type t_2. The case variables $a_1 \ldots a_{jK}$ are free in the case term u_j.

$$\begin{aligned}&[\![\textbf{prec}\ d : t_1 \rightarrow t_2 [\!]_j\, c_j\ a_{j1} \ldots a_{jK} \Longrightarrow u_j]\!]_\rho = \textbf{prec}\ [\![t_1]\!]_\rho \rightarrow [\![t_2]\!]_\rho [\!]_j [\![c_j]\!]_{\rho'}\ K_j \Longrightarrow [\![u_j]\!]_{\rho_j} \\ &\quad \textbf{where}\ \rho' \ = \ \rho + d \\ &\qquad\qquad \rho_j \ = \ \rho + a_{j1} + \ldots + a_{jK}\end{aligned}$$

10.3.3 Type Correctness

The primitive recursive term ($\textbf{prec}\ t_1 \rightarrow t_2 [\!]_j\, l.c_j\ K_j \Longrightarrow u_j$) must satisfy the following rules:

(i) The term t_1 must be a datatype, with an actual parameter for each formal parameter of the datatype declaration.

(ii) The body of the recursion must have one case for each case of the datatype – in any order. The case identifier c_j must be a constructor of the datatype.

(iii) The case variables $\underline{0 \ldots K_j - 1}$ must correspond to the accessors of the datatype case constructed by c_j.

Example To understand the type checking in more detail, we consider the example shown on page 31. The alternative `cons h t ==> h + t` introduces two variables. The variable h stands for the *head* of the list; its type is the list element type. The case variable t is treated differently since it corresponds to the recursive accessor of the list; instead of standing for the *tail* of the list, it stands for the recursive application of the function, i.e. *sum_max* applied to the *tail* of the list. The type of t is therefore the range type of the primitive recursion, declared as $Upto\ (max \times length\ l)$. This type depends on the list l to which the recursive function is applied: so that this type can be constructed correctly, a second bound variable t' is introduced, which *does* stand for the tail of the list. This is summarised in the following table:

Variable	Type
h	$Upto\ max$
t'	$ListM\ max$
t	$Upto\ (max \times length\ t')$

The types of h, t' and t form the context for type checking the term $h + t$.

10.3.3.1 Type Checking

The *type_ok* function for the primitive recursive term is specified by:

$$
\begin{array}{l}
type_ok_\rho(\mathbf{prec}\ t_1 \rightarrow t_2 \|_j\ l.c_j\ K_j \Longrightarrow u_j) = \\
\quad type_ok_\rho\, t_1 \wedge type_inf_\rho\, t_1 = * \wedge t_1 \rightsquigarrow d\ p_1 \ldots p_I \wedge is_adt_\rho\, d\ \wedge \\
\quad type_ok_{\rho'}\, t_2 \wedge \mathbf{kind_or_type}(type_inf_{\rho'}\, t_2)\ \wedge \\
\quad (\forall j, j' \in 1 \ldots J \cdot j \neq j' \Rightarrow l.c_j \neq l.c_{j'})\ \wedge \\
\quad \{\ l.c_j | j \in 1 \ldots J\ \} = adt_cons_ids_{\rho'}\, d\ \wedge \\
\quad \bigwedge_{j=1}^{J}(type_ok_{\rho^j}\, u_j^{+R_j} \wedge type_inf_{\rho^j}\, u_j^{+R_j} \approx t_2^{+(R_j+K_j)}\ \wedge \\
\qquad K_j = adt_num_accs_{\rho'}\, l.c_j\ p_1 \ldots p_I)
\end{array}
$$

where

$$
\begin{array}{ll}
\rho' & = \rho + type_inf_\rho\, t_1 \\
R_j & = \text{number of recursive datatype components in case } j \\
\rho^j & = rec_case_env\ \rho'\ (adt_cons_type_{\rho'}\, l.c_j\ \langle p_1^+, \ldots, p_I^+ \rangle)\ t_2\ 0
\end{array}
$$

The environment for type checking the case term u_j is determined from the type of the constructor for the case, using the *rec_case_env* function. The arguments of this function are:

(i) the initial environment ρ'

(ii) the constructor type, given by the $adt_cons_type_\rho$ function

(iii) the range type of the recursive function, which forms the type of a case variable which corresponds to a recursive element of the datatype

(iv) the number of case variables added to the environment, counting two for a recursive element of the datatype, initially 0.

The function *rec_case_env* is specified recursively by the following equations:

$$\begin{array}{lcl} rec_case_env\ \rho\ C\ t\ n & = & rec_case_env(\rho + A)\ C'\ t^+\ (n+1) \\ & & \textbf{where } C = (\Pi A \cdot C') \text{ and } A \text{ is not recursive} \\ rec_case_env\ \rho\ C\ t\ n & = & rec_case_env(\rho + A + t^+[n+1 := \underline{0}])\ C'\ t^{+2}\ (n+2) \\ & & \textbf{where } C = (\Pi A \cdot C') \text{ and } A \text{ is recursive} \\ rec_case_env\ \rho\ C\ t\ n & = & \rho \textbf{ otherwise} \end{array}$$

10.3.3.2 Type Inference

The type of primitive recursive term is taken directly from the syntax.

$$type_inf_\rho(\textbf{prec } t_1 \rightarrow t_2 \underset{j}{[\!]}\, l.c_j\ K_j \Longrightarrow u_j) = (\Pi t_1 \cdot t_2)$$

10.3.3.3 Type Correctness Conditions

Context In each case, the datatype variable is equal to the term constructed by the case constructor and the case variables. This equality is used as a context term in the type correctness condition. In particular, this context term constrains the additional bound variable introduced for each recursive accessor. The context terms for the two cases of the `sum_max` example given above are shown in the following table:

Constructor	Context
nil	$l = nil$
$cons$	$l = cons\ h\ t'$

TCC's The function *tcc* for the primitive recursive term, giving the type correctness conditions, is specified by:

$$\begin{array}{l} tcc_\rho(\textbf{prec}\ t_1 \rightarrow t_2 [\!]_j\ l.c_j\ K_j \Longrightarrow u_j) = \\ \quad tcc_\rho\, t_1 \wedge (\forall t_1 \cdot tcc_{\rho'}\, t_2) \wedge \\ \quad \bigwedge_j (\forall t_1 \cdot rec_case_forall(adt_cons_type_{\rho'}\, c_j\ \langle p_1^+\ \ldots\ p_I^+\rangle)\ t_2\ 0\ l.c_j \\ \qquad (tcc_{\rho^j}\, u_j^{+R_j} \wedge inject_type_{\rho^j}\, u_j^{+R_j}\ t_2^{+(R_j+K_j)}) \\ \textbf{where} \\ \quad \rho' = \rho + type_inf_\rho\, t_1 \\ \quad R_j = \text{number of recursive datatype components in case } j \\ \quad t_1 \rightsquigarrow d\ p_1\ \ldots\ p_I \\ \quad \rho^j = rec_case_env\ \ \rho'\ (adt_cons_type_{\rho'}\, l.c_j\ \langle p_1^+, \ldots, p_I^+\rangle)\ t_2\ 0 \end{array}$$

The function *rec_case_forall* provide the context and the quantification over the case variables for the type correctness condition for each case term. The arguments of this function are:

(i) the constructor type, given by the $adt_cons_type_\rho$ function

(ii) the range type of the recursive function, which forms the type of a case variable which corresponds to a recursive element of the datatype

(iii) the number of case variables added to the environment, counting two for a recursive element of the datatype, initially 0

(iv) the constructor term for the case, and

(v) the type correctness condition for injecting the case term into the range type.

The function *rec_case_forall* is specified recursively by the following equations:

$$\begin{array}{rcl} rec_case_forall\, C\ t\ n\ x\ p & = & (\forall A \cdot rec_case_forall\ \ C'\ t^+\ (n+1)\ (x^+\ \underline{0})\ p) \\ & & \textbf{where } C = (\Pi A \cdot C') \textbf{ and } A \text{ is not recursive} \\ rec_case_forall\, C\ t\ n\ x\ p & = & (\forall A \cdot (\forall t^+[n+1 := \underline{0}] \cdot \\ & & \qquad rec_case_forall\ \ C'\ t^{+2}\ (n+2)\ (x^{+2}\ \underline{1})\ p)) \\ & & \textbf{where } C = (\Pi A \cdot C') \textbf{ and } A \text{ is recursive} \\ rec_case_forall\, C\ t\ n\ x\ p & = & (\underline{n} = x) \Rightarrow p\ \textbf{otherwise} \end{array}$$

Each part A of the constructor type C is the type of one element of the datatype case, corresponding to one of the bound variables in the recursion term. The type correctness condition is formed by universal quantification over the bound variable types of the type correctness conditions for each case of the datatype, in the appropriate context.

10.3.3.4 Injecting a Term into a Type

Since the primitive recursion term is not used to form a type, a new case of the type injecting function *inject_type* is not needed – the default case applies.

10.3.3.5 Reduction

A primitive recursion function applied to a datatype element with an particular constructor can be reduced. Let:

$$Rec = (\mathbf{prec}\ D\ p_1 \ldots p_I \rightarrow t \,[\!]_j\, l.c_j\ K_j \Longrightarrow u_j)$$
$$A = l.c_i\ p_1 \ldots p_I\ t_K \ldots t_R \ldots t_1$$

where $t_k \ldots t_1$ are terms and the Rth accessor of the ith case is recursive. Then:

$$Rec\ A \rightsquigarrow (u_i[K-1 := t_K^{+K}] \ldots [R-1 := (Rec\ t_R)^{+K}] \ldots [0 := t_1^{+K}])^{-K}$$

10.3.4 Type Errors

A **prec** term t is not well-typed when the $type_ok_p\, t$ is false. The type errors are summarised in the following table, showing the conjunct of $type_ok_p$ which is falsified for each error:

$type_ok$ conjunct	Error
$type_inf_p\, t_1 = * \wedge t_1 \rightsquigarrow d\ p_1 \ldots p_I \wedge is_adt_p\, d$	t_1 is not a parameterised datatype
$\mathbf{kind_or_type}(type_inf_{p'}\, t_2)$	t_2 is not a type or kind
$(\forall j, j' \in 1 \ldots J. j \neq j' \Rightarrow l.c_j \neq l.c_{j'})$	repeated constructor
$\{\ l.c_j \mid j \in 1 \ldots J\ \} = adt_cons_ids_{p'}\, d$	missing or invalid constructor
$K_j = adt_num_accs_{p'}\, l.c_j\ p_1 \ldots p_I$	wrong number of case variables

Type Compatibility A case for the type compatibility relation $\approx$ is required for the **prec** term, even though this term never represents a type (see page 131). The specification is:

$$(\text{Prec})\quad \frac{u_j \approx u'_j \quad t_1 \approx t'_1 \quad t_2 \approx t'_2}{(\mathbf{prec}\ t'_1 \rightarrow t'_2 \,[\!]_j\, l.c_j\ K_j \Longrightarrow u'_j) \approx (\mathbf{prec}\ t_1 \rightarrow t_2 \,[\!]_j\, l.c_j\ K_j \Longrightarrow u_j)}$$

10.4 General Recursion

The general recursion term allows a function to be constructed recursively, using a measure function to show that the recursion terminates.

10.4.1 Abstract Syntax

The abstract syntax with identifiers for the general recursion term is:

$$\mathcal{T}_x ::= \ldots \mid (\mu\mathcal{I} : \mathbf{seq}(\mathcal{I} : \mathcal{T}_x) \rightarrow \mathcal{T}_x | \mathcal{T}_x . \mathcal{T}_x)$$

In the term $(\mu f : \langle x_0 : t_1, x_1 : t_2\rangle \to u|m.b)$ the terms t_1, t_2 are the domain types of the functions and u is the range type. The measure is m and b is the body. The bound variables in the term are f, the variable which stands for the function being defined and x_j, the variables standing for the function arguments.

10.4.2 Visibility of Identifiers

The variable-free version of the syntax is:

$$\mathcal{T}_i ::= \ldots \mid (\mu\,\mathbf{seq}\,\mathcal{T}_i \to \mathcal{T}_i|\mathcal{T}_i.\mathcal{T}_i)$$

Free Variables The free variables of the general recursion term are specified by:

$$\begin{array}{rcl}\Phi(\mu\langle t_j\rangle_{j=0}^{j=N-1} \to u|m.b) & = & \bigcup_{j=0}^{N-1}\{\ i-j \mid i \in \Phi\, t_j \wedge i \geq j\ \} \cup \\ & & \{\ i-(N-1) \mid i \in \Phi\, u \cup \Phi\, m \wedge i \geq N-1\ \} \cup \\ & & \{\ i-N \mid i \in \Phi\, b \wedge i \geq N\ \}\end{array}$$

Encoding The encoding of the general recursion term in the syntax with indices is specified by:

$$\begin{array}{l}[\![(\mu f : \langle x_j : t_j\rangle_{j=0}^{j=n-1} \to u|m.b]\!]_\rho = (\mu\langle [\![t_j]\!]_{\rho_i}\rangle_{j=0}^{j=n-1} \to [\![u]\!]_{\rho_n}|[\![m]\!]_{\rho_n}.[\![b]\!]_{\rho'}) \\ \quad \begin{array}{lcl}\textbf{where } \rho_0 & = & \rho \\ \rho_{j+1} & = & \rho + x_j \textbf{ for } j = 0, \ldots, n-1 \\ \rho' & = & \rho_n + f\end{array}\end{array}$$

The recursive variable f is only free in the body of the recursion b. An additional check is required to ensure that all occurences of f are fully parameterised (i.e. there are N parameters).

10.4.3 Type Correctness

10.4.3.1 Type Checking

The term $(\mu\langle t_j\rangle_{j=0}^{j=N-1} \to u|m.b)$ is well-typed if:

(i) each of the terms t_j and u are well-typed and can be used as a type, and

(ii) the measure term m is well-typed and has the type $\mathbb{N}$, and

(iii) the term in the body of the recursion has the declared type u, assuming that the recursive function has the type $(\Pi t_0.(\Pi t_1.\ldots.u))$.

The *type_ok* function for the general recursion term is specified by:

$$
\begin{aligned}
& type_ok_{\rho}(\mu\langle t_j\rangle \rightarrow u|m.b) = \\
& \quad \textstyle\bigwedge_j (type_ok_{\rho_j}\ t_j \wedge \mathbf{kind_or_type}(type_inf_{\rho_j}\ t_j)) \wedge \\
& \quad type_ok_{\rho_n}\ u \wedge \mathbf{kind_or_type}(type_inf_{\rho_n}\ u) \wedge \\
& \quad type_ok_{\rho_n}\ m \wedge type_inf_{\rho_n}\ m \equiv \mathbb{N} \wedge \\
& \quad type_ok_{\rho'}\ b \wedge \underline{0} \notin \Phi\ (type_inf_{\rho'}\ b) \wedge (type_inf_{\rho'}\ b)^{-} \approx u \\
\textbf{where}\ & \rho_0 = \rho \\
& \rho_{j+1} = \rho_j + t_j \text{ for } j = 0, \ldots, n-1 \\
& \rho' = \rho_n + (\Pi t_0. \ldots (\Pi t_{n-1}.u)\ldots)^{+n}
\end{aligned}
$$

Note that the term b is in an environment containing the recursive variable: an additional check is required to ensure that this variable is not free in the type of b, before the type of b is checked for compatibility with u. The requirement for the measure term m to have type $\mathbb{N}$ ensures that the measure has a minimum.

10.4.3.2 Type Inference

The type of the recursive term made explicit by the domain and range types. The specification of the *type_inf* function is:

$$type_inf_{\rho}(\mu\langle t_j\rangle_0^{n-1} \rightarrow u|m.b) = (\Pi t_0. \ldots (\Pi t_{n-1}.u)\ldots)$$

10.4.3.3 Type Correctness Conditions

The type correctness condition for the recusive term is given by the conjunction of the type correctness conditons for the sub-terms.

$$
\begin{aligned}
tcc_{\rho}(\mu\langle t_i\rangle_0^{n-1} \rightarrow u|m \cdot b) \;=\; & \textstyle\bigwedge_{i=0}^{i=n-1} (\forall t_0 \cdot \ldots (\forall t_{i-1} \cdot tcc_{\rho_i}\ t_i)\ldots) \wedge \\
& (\forall t_0 \cdot \ldots (\forall t_{n-1} \cdot tcc_{\rho_n}\ u \wedge tcc_{\rho_n}\ m)\ldots) \wedge \\
& (\forall t_0 \cdot \ldots (\forall t_{n-1} \cdot (\forall F \cdot \\
& \qquad tcc_{\rho'}\ b \wedge inject_type_{\rho'}\ b\ u^{+}))\ldots) \\
\textbf{where}\ \rho_0 \;=\; & \rho \\
\rho_{i+1} \;=\; & \rho_i + t_i \text{ for } i = 0, \ldots, n-1 \\
F \;=\; & (\Pi t_0{}^{+n} \cdot \ldots (\Pi\{\ (\lambda t_{n-1}{}^{+n,n-1} \cdot m^{+n,n} < m^{+n})\ \} \cdot u^{+n,n})\ldots) \\
\rho' \;=\; & \rho_n + F
\end{aligned}
$$

The type correctness conditon for the body of the recursion b is a special case. Firstly, the type of this term is given explicitly, so a type injection is used to check that the type is correct. Secondly, the type given to the recursive variable is a subtype, formed to ensure that the parameters to each recursive call of the function decrease the variant.

10.4.3.4 Injecting a Term into a Type

Since the general recursion term is not used to form a type, a new case of the type injecting function *inject_type* is not needed – the default case applies.

10.4.3.5 Reduction

A reduction for the recursion term corresponding to a single syntactic unrolling of the recursive definition can be specified. Let $Rec = (\mu\langle t_i\rangle_0^{n-1} \rightarrow u|m.b)$, then:

$$Rec \rightsquigarrow (\lambda t_0. \ldots (\lambda t_{n-1}.b[n := Rec^{+n}]) \ldots)$$

Since the 'reduced' form includes the original function, there is no finite reduced syntactic form of a recursion term *Rec* which does not include *Rec*. Since the measure decreases on each application to a recursion term, the fact that reduction does not terminate might be taken to indicate that the reduction is not well-typed. We consider that this is not the case since after sufficient reductions to reduce the measure function below zero, the recursive call occurs in the false context: anything is well-typed in this context.

10.4.4 Type Errors

A μ term t is not well-typed when the $type_ok_\rho\ t$ is false. The type errors are summarised in the following table, showing the conjunct of $type_ok_\rho$ which is falsified for each error:

type_ok conjunct	Error
$\mathbf{kind_or_type}(type_inf_{\rho_j}\ t_j)$	t_j is not a type or kind
$\mathbf{kind_or_type}(type_inf_{\rho_n}\ u)$	u is not a type or kind
$type_inf_{\rho_n}\ m \equiv \mathbb{N}$	the measure m must have type $\mathbb{N}$
$\underline{0} \notin \Phi\ (type_inf_{\rho'}\ b)$	the function being defined may not be free in the type of the body b of the recursion
$(type_inf_{\rho'}\ b)^- \approx u$	the type of the body b of the recursion is not compatible with the annotated type u

Type Compatibility A case for the type compatibility relation $\approx$ is required for the μ term, even those this term never represents a type (see page 131). The specification is:

$$\text{(Recursion)}\ \frac{t_i \approx t_i' \quad m \approx m' \quad b \approx b'}{(\mu\langle t_i\rangle_0^{n-1} \rightarrow u|m.b) \approx (\mu\langle t_i'\rangle_0^{n-1} \rightarrow u'|m'.b')}$$

Chapter 11

Statements

This chapter describes the statements of the language. The language is described using an abstract syntax in Section 11.1. The visibility of identifiers is described in Section 11.2 and well-formation and type checking are considered in Sections 11.3, 11.4 and 11.5. These are followed by sections describing the meaning of statements. Section 11.10 describes the propagation of context assertions in statements.

11.1 Abstract Syntax

Statements belong to the syntactic category $\mathcal{S}_x$ of statements. Variables are indicated by x if they stand for an ordinary value, or by X if they stand for a statement.

$$
\begin{array}{lcll}
\mathcal{S} & ::= & \mathbf{skip} & \text{skip} \\
 & \mid & \mathbf{abort} & \text{abort} \\
 & \mid & \mathcal{L}_1, \ldots, \mathcal{L}_n := \mathcal{T}_1, \ldots, \mathcal{T}_n & \text{multiple asgn.} \\
 & \mid & x_1, \ldots, x_n := \mathbf{where}\ \mathcal{T} & \text{where asgn.} \\
 & \mid & \mathcal{L}_1, \ldots, \mathcal{L}_n := \mathbf{any}\ x_1, \ldots, x_n\ \mathbf{where}\ \mathcal{T} & \text{abstract asgn.} \\
 & \mid & (\mathbf{pre}\ \mathcal{T}\ \mathbf{in}\ \mathcal{S}) & \text{precondition} \\
 & \mid & \mathcal{S}; \mathcal{S}' & \text{seq. comp.} \\
 & \mid & (\mathbf{if}\ \mathbf{or}_i\ \mathcal{T}_i \Longrightarrow \mathcal{S}_i) & \text{conditional} \\
 & \mid & (\mathbf{while}\ \mathcal{T}\ \mathbf{vary}\ \mathcal{T}\ \mathbf{loop}\ \mathbf{or}_i\ \mathcal{T}_i \Longrightarrow \mathcal{S}_i) & \text{loop} \\
 & \mid & (\mathbf{var}\ x : \mathcal{T} | \mathcal{T}\ \mathbf{in}\ \mathcal{S}) & \text{variable} \\
 & \mid & (\mathbf{con}\ x \widehat{=} \mathcal{T}\ \mathbf{in}\ \mathcal{S}) & \text{logical constant} \\
 & \mid & X & \text{call} \\
 & \mid & (\mathbf{proc}\ X \widehat{=} \mathcal{S}\ \mathbf{in}\ \mathcal{S}') & \text{procedure} \\
 & \mid & [\ \mathcal{A}, \ldots, \mathcal{A}'\ ]\ \mathcal{S} & \text{abstraction} \\
 & \mid & \mathcal{S}\ [\ \mathcal{T}, \ldots, \mathcal{T}'\ ] & \text{application} \\
 & \mid & (\mathbf{rec}\ X \widehat{=} \mathcal{S}\ \mathbf{vary}\ \mathcal{T}) & \text{recursion}
\end{array}
$$

Variable Initialisation All variables must be initialised. The abstract syntax for a variable initialisation includes a term which is the initial value. The initial value can be specified by a predicate by use a choice term. In the concrete syntax, a shorter form is provided to specify the initial value by a predicate. The initialisation may also be omitted: in this case the variable is initialised to an arbitrary value *within its type*. The abstract representation of the different forms in the concrete syntax is as follows:

Concrete	Abstract
`VAR x:T := e IN S END VAR`	$(\textbf{var}\ x : t \mid e\ \textbf{in}\ S)$
`VAR x:T := WHERE p IN S END VAR`	$(\textbf{var}\ x : T \mid (\epsilon\, x : T.p)\ \textbf{in}\ S)$
`VAR x:T IN S END VAR`	$(\textbf{var}\ x : T \mid (\epsilon\, x : T.\mathbf{t})\ \textbf{in}\ S)$

Recursion In the abstract syntax $\mathcal{S}_x$ (and the concrete syntax) the recursion block contains only a single statement S. Two steps are required to introduce a recursive implementation:

(i) introduce a recursion statement containing a statement S which does not use the recursive variable,

(ii) refine this statement to another T, which may reference the recursive variable.

In this way, the statement S initially acts as the specifying statement of the recursion. In the next section, we formalise the requirement that the recursive variable must not be used in S. The recursion block is the only statement in the language described here which cannot be used without introducing refinement steps.

Parameterisation Mechanisms The syntactic category $\mathcal{A}$ contains the four parameterisation mechanism which can be used in the abstraction statement:

$$\mathcal{A} ::= \textbf{val}\, x : \mathcal{T} \mid \textbf{res}\, x : \mathcal{T}\ \mathcal{T} \mid \textbf{valres}\, x : \mathcal{T} \mid \textbf{ref}\ x : \mathcal{T}$$

A **val** parameter has copy-in semantics and a **res** parameter has copy-out; the initial value of a result parameter is made explicit, in the same way as in a variable declaration. The value-result parameter **valres** combines copy-in and copy-out. A reference parameter **ref** can also be used to pass a value both into and out of a parameterised statement.

Left Values The syntactic category $\mathcal{L}$ contains the subset of $\mathcal{T}$ which can be the left-hand side of an assignment statement.

$$\begin{array}{rcll} \mathcal{L} & ::= & x & \text{variable} \\ & \mid & \mathcal{L}\ \mathcal{T} & \text{function application} \\ & \mid & a\ \mathcal{L} & \text{datatype accessor} \\ & \mid & \mathcal{L}\,\hat{}\,\mathcal{T} & \text{map index} \end{array}$$

The operator ˆ selected an element from a map; it is declared in the *Map* theory on page 214. The three compound forms of left value are all forms of application; however, a more elaborate abstract syntax is used to clarify the differences between the forms.

Example Left Values Assuming that T and M are types, then with the following type declarations ($Array$[1] is declared on page 215):

$$
\begin{array}{ll}
A == Array\ 10\ T & AA == Array\ 10\ A \\
M == Map\ T\ T & AM == Array\ 10\ M \\
\multicolumn{2}{l}{D ::= c_one\ (one : T)\ (two : A)\ (three : AM)\ is_one} \\
AD == Array\ 10\ D &
\end{array}
$$

let $a : A$, $aa : AA$, $m : M$, $d : D$, $am : AM$ and $ad : AD$ be variables and t be a term of type T then the following terms are left values:

$$
\begin{array}{lll}
a\ 1 & aa\ 2 & aa\ 2\ 3 \\
m\,\hat{}\,t & am\ 1 & am\ 1\,\hat{}\,t \\
one\ d & two\ d\ 1 & (three\ d\ 1)\,\hat{}\,t \\
one\ (ad\ 1) & three\ (ad\ 1)\ 2\,\hat{}\,t &
\end{array}
$$

Any left value can also be parsed as a term: in the specification we assume conversion from left values to terms when required.

11.2 Visibility of Identifiers

Following the approach of Section 7.3.1, an alternative syntax $\mathcal{S}_i$ is defined using DeBruijn indices. Figure 11.1 show the abstract syntax for cases where $\mathcal{S}_x$ and $\mathcal{S}_i$ differ, with i or I standing for an index. Note that the two forms of abstract assignment in $\mathcal{S}_x$ are represented by a single form in $\mathcal{S}_i$, using the equivalence noted on page 37.

The translation of $s \in \mathcal{S}$ into $\mathcal{S}_i$ is $[\![s]\!]_\rho$ where ρ is the visibility environment of Section 8.1.1. The transformation determines the visibility rules of the identifiers, since a term which contains undeclared identifiers cannot be encoded. Note that the ordinary variables x and the statement variables X belong to the same namespace. The encoding function is specified in Figure 11.2.

Recursion In the abstract syntax $\mathcal{S}_i$, a recursion statement has *two* statements. In (**rec** S **vary** E **in** T_X), X is the *recursive statement variable*, S is a statement which gives the meaning of a recursive call, while T_X is the statement in which the recursive call may be made. The variant expression, which must decrease before each call statement is E; note that the initial value of the variant can be referred to as $vary_X$[2].

[1] An array type is a form of function. Any function type can be used in a left value, not just arrays.

[2] It would be more consistent with the rest of the language to allow the user to choose the variant identifier. Since it is not often needed, the syntax should allow it to be omitted.

$$
\begin{array}{lcll}
\mathcal{L}_i & ::= & \underline{i} & \text{variable} \\
& | & \mathcal{L}_i\ \mathcal{T}_i & \text{function application} \\
& | & h.a\ \mathcal{L}_i & \text{datatype accessor} \\
& | & \mathcal{L}_i\ \hat{}\ \mathcal{T}_i & \text{map index}
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{S}_i & ::= & \ldots & \\
& | & \mathcal{L}_1, \ldots, \mathcal{L}_n := \mathcal{T}_{i1}, \ldots, \mathcal{T}_{in} & \text{multiple asgn.} \\
& | & \mathcal{L}_1, \ldots, \mathcal{L}_n := \mathbf{any}\ \mathbb{N}\ \mathcal{T}_i & \text{abstract asgn.} \\
& | & (\mathbf{var}\ \mathcal{T}_i | \mathcal{T}_i\ \mathbf{in}\ \mathcal{S}_i) & \text{variable} \\
& | & (\mathbf{con}\ \mathcal{T}_i\ \mathbf{in}\ \mathcal{S}_i) & \text{logical constant} \\
& | & \underline{I} & \text{call} \\
& | & (\mathbf{proc}\ \mathcal{S}_i\ \mathbf{in}\ \mathcal{S}'_i) & \text{procedure} \\
& | & [\ \mathcal{A}_i, \ldots, \mathcal{A}'_i\ ]\ \mathcal{S}_i & \text{abstraction} \\
& | & (\mathbf{rec}\ \mathcal{S}\ \mathbf{vary}\ \mathcal{T}\ \mathbf{in}\ \mathcal{S}') & \text{recursion}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{A}_i & ::= & \mathbf{val}\,\mathcal{T}_i \mid \mathbf{res}\,\mathcal{T}_i\ \mathcal{T}_i \\
& | & \mathbf{valres}\,\mathcal{T}_i \mid \mathbf{ref}\ \mathcal{T}_i
\end{array}
$$

Figure 11.1: Abstract Syntax with Indices

This construct represents a recursive statement because the statement T_X is a refinement of S; in fact S *is* the least refined part of T_X; this is expressed using the function *spec* which is described in Chapter 12. It is important to remember that the two statements in the recursion are not independent.

Abstraction The encoding of an abstraction $\mathbf{val}\,x\ t$ is represented by $[\![\mathbf{val}\,x\ t]\!]_\rho^{\rho'}$ where ρ is the initial environment and ρ' is the environment updated with the formal parameter identifier.

$$
\begin{array}{lcl}
[\![\mathbf{val}\,x\ t]\!]_\rho^{\rho+x} & = & \mathbf{val}[\![t]\!]_\rho \\
& & \mathbf{provided}\ x \notin \mathbf{dom}\,\nu\ \mathbf{where}\ \rho = \iota, \delta, \nu \\
[\![\mathbf{res}\,x\ t\ t_0]\!]_\rho^{\rho+x} & = & \mathbf{res}[\![t]\!]_\rho\ [\![t_0]\!]_\rho \\
& & \mathbf{provided}\ x \notin \mathbf{dom}\,\nu\ \mathbf{where}\ \rho = \iota, \delta, \nu
\end{array}
$$

The encoding of modes **valres** and **ref** is similar to that of **val**.

Parameterised Recursion A refinement statement can be parameterised. Consider the following example, which illustrates the two ways actual parameters can be associated with recursion. A recursion block is introduced to refine the specification `f := fact n`, where `f` and `n` are integer variables. The recursion block is parameterised with formal parameters `x` and `g`. The variables `n` and `f` are used as actual parameters

$$
\begin{array}{lcl}
[\![\textbf{skip}]\!]_\rho & = & \textbf{skip} \\
[\![\textbf{abort}]\!]_\rho & = & \textbf{abort} \\
[\![l_1, \ldots, l_m := t_1, \ldots, t_n]\!]_\rho & = & [\![l_1]\!]_\rho, \ldots, [\![l_m]\!]_\rho := [\![t_1]\!]_\rho, \ldots, [\![t_n]\!]_\rho \\
[\![x_1, \ldots, x_m := \textbf{where } t]\!]_\rho & = & [\![x_1]\!]_\rho, \ldots, [\![x_m]\!]_\rho := \textbf{any } m \ [\![t]\!]_{\rho+x_1+\ldots+x_m} \\
\multicolumn{3}{l}{[\![l_1, \ldots, l_m := \textbf{any } y_1, \ldots, y_n \textbf{ where } t]\!]_\rho} \\
 & = & [\![l_1]\!]_\rho, \ldots, [\![l_m]\!]_\rho := \textbf{any } n \ [\![t]\!]_{\rho+y_1+\ldots+y_n} \\
 & & \textbf{provided } y_1 \ \ldots \ y_n \text{ are distinct} \\
[\![(\textbf{pre } t \textbf{ in } s)]\!]_\rho & = & (\textbf{pre } [\![t]\!]_\rho \textbf{ in } [\![s]\!]_\rho) \\
[\![s_1 ; s_2]\!]_\rho & = & [\![s_1]\!]_\rho ; [\![s_2]\!]_\rho \\
[\![\textbf{if or}_i \ t_i \Longrightarrow s_i]\!]_\rho & = & \textbf{if or}_i \ [\![t_i]\!]_\rho \Longrightarrow [\![s_i]\!]_\rho \\
\multicolumn{3}{l}{[\![\textbf{while } t_I \textbf{ vary } t_v \textbf{ loop or}_i \ t_i \Longrightarrow s_i]\!]_\rho} \\
 & = & \textbf{while } [\![t_I]\!]_\rho \textbf{ vary } [\![t_v]\!]_\rho \textbf{ loop or}_i \ [\![t_i]\!]_\rho \Longrightarrow [\![s_i]\!]_\rho \\
[\![(\textbf{var } x : t | e \textbf{ in } s)]\!]_\rho & = & (\textbf{var } [\![t]\!]_\rho | [\![e]\!]_\rho \textbf{ in } [\![s]\!]_{\rho+x}) \\
 & & \textbf{provided } x \notin \textbf{dom}\, \nu \textbf{ where } \rho = \iota, \delta, \nu \\
[\![(\textbf{con } x \widehat{=} e \textbf{ in } s)]\!]_\rho & = & (\textbf{con } [\![e]\!]_\rho \textbf{ in } [\![s]\!]_{\rho+x}) \\
 & & \textbf{provided } x \notin \textbf{dom}\, \nu \textbf{ where } \rho = \iota, \delta, \nu \\
[\![X]\!]_\rho & = & \underline{\nu \ X} \textbf{ provided } X \in \textbf{dom}\, \nu \textbf{ where } \rho = \iota, \delta, \nu \\
[\![(\textbf{proc } X \widehat{=} s_1 \textbf{ in } s_2)]\!]_\rho & = & (\textbf{proc } [\![s_1]\!]_\rho \textbf{ in } [\![s_2]\!]_{\rho+X}) \\
 & & \textbf{provided } X \notin \textbf{dom}\, \nu \textbf{ where } \rho = \iota, \delta, \nu \\
[\![[\ a_1, \ldots, a_n \ ]\ s]\!]_\rho & = & [\ [\![a_1]\!]_\rho^{\rho_1}, \ldots, [\![a_n]\!]_{\rho_{n-1}}^{\rho_n} \ ]\ [\![s]\!]_{\rho_n} \\
[\![s\ [\ t_1, \ldots, t_n \ ]]\!]_\rho & = & [\![s]\!]_\rho \ [\ [\![t_1]\!]_\rho, \ldots, [\![t_n]\!]_\rho \ ] \\
[\![(\textbf{rec } X \widehat{=} s \textbf{ vary } t)]\!]_\rho & = & (\textbf{rec } s'' \textbf{ vary } [\![t]\!]_\rho \textbf{ in } s') \\
 & & \textbf{where } s' = [\![s]\!]_{\rho+vary_X+X} \\
 & & \qquad\quad s'' = (spec\, s')^{-2} \\
 & & \textbf{provided } \{0, 1\} \cap \Phi(spec\, s') = \emptyset \\
 & & \qquad\qquad \{vary_X, X\} \cap \textbf{dom}\, \nu = \emptyset \\
 & & \qquad\qquad\quad \textbf{where } \rho = \iota, \delta, \nu
\end{array}
$$

Figure 11.2: Syntax Transformation for Statements

of the recursion block. A recursive call is used in the refine the specifying statement of the recursion. The actual parameters of the call are `x - 1` and `g`.

```
REF 1 BY
  REC Fact IS [VAL x: Nat; RES g: Nat]
    PRE  0 <= x IN
      <<2>> g := fact x
    END PRE
  VARY x END REC [n, f]
END
```

```
REF 2 BY
  IF x = 0 ==> g := 1
  OR x > 0 ==> Fact[x - 1, g];
               g := g * x
  END IF
END
```

As shown in this example, the formal parameters of the recursion may be free in the variant[3] (here the variant is x, one of the formal parameters). The encoding of parameterised recursion requires a separate case[4]. Note that the formal parameters are free in both the component statements of the recursion.

$$
\begin{array}{l}
[\![(\mathbf{rec}\ X \widehat{=} [\ a_1, \ldots, a_n\ ]\ s\ \mathbf{vary}\ t)]\!]_{\rho_0} = \\
\qquad (\mathbf{rec}\ [\ b_1, \ldots, b_n\ ]\ s''\ \mathbf{vary}\ [\![t]\!]_{\rho_n}\ \mathbf{in}\ s') \\
\qquad \mathbf{where}\ b_i = [\![a_i]\!]^{\rho_i}_{\rho_{i-1}} \\
\qquad\qquad s' = [\![s]\!]_{\rho_n + vary_X + X} \\
\qquad\qquad s'' = (spec\ s')^{-2} \\
\qquad \mathbf{provided}\ \{0, 1\} \cap \Phi(spec\ s') = \emptyset \\
\qquad\qquad \{vary_X, X\} \cap \mathbf{dom}\,\nu = \emptyset\ \mathbf{where}\ \rho_n = \iota, \delta, \nu
\end{array}
$$

This case generalises that for recursion without parameters; we specify some functions only for a parameterised recursion, when the specification for the simpler case can be inferred.

Redeclaration of Identifiers Side-conditions of the equations in Figure 11.2 specify the following well-formation rules associated with variables:

(i) the identifier used for a variable, formal parameter, constant or a statement variable may not already have a declaration, and

(ii) the identifiers used for the bound variables in an abstract assignment must be different.

Re-declaration of a variables is prohibited in Euclid [LGH+78] and SPARK [CJM+92, CGM92] to prevent variable capture in verification conditions. Since the use of De Bruijn indices avoids variable capture, the first of these rules is not necessary. However, unique declaration may still be considered a useful restraint, avoiding potential confusion: see [WSH81] for a description of the ambiguity introduced into Pascal as a result of permitting the redeclaration of types. Preventing re-declaration has the disadvantage that a well-formed program can become ill-formed when placed in a different context.

11.2.1 Free Variables and Substitution

Following the approach of Section 9.3 for terms, functions for the free variables of statements and for the substitution of variables in statements are introduced.

[3]This is the reason the variant is written at the end of the refinement statement.

[4]The factorisation of the language with a separate abstraction statement is not satisfactory for parameterised recursion.

$$
\begin{array}{lcl}
(l_1,\ldots,l_m := t_1,\ldots,t_n)[i := u] & = & l_1[i := u],\ldots,l_m[i := u] := \\
 & & \quad t_1[i := u],\ldots,t_n[i := u] \\
(l_1,\ldots,l_m := \textbf{any } n\ t)[i := u] & = & l_1[i := u],\ldots,l_m[i := u] := \textbf{any } n\ t[i+n := u^{+n}] \\
(\textbf{var } t_1|t_2 \textbf{ in } S)[i := u] & = & (\textbf{var } t_1[i := u]|t_2[i := u] \textbf{ in } S[i+1 := u^{+}]) \\
(\textbf{con } t \textbf{ in } S)[i := u] & = & (\textbf{con } t[i := u] \textbf{ in } S[i+1 := u^{+}]) \\
I[i := u] & = & u \quad \textbf{if } i \equiv I \\
 & = & I \quad \textbf{otherwise} \\
(\textbf{proc } S_1 \textbf{ in } S_2)[i := u] & = & (\textbf{proc } S_1[i := u] \textbf{ in } S_2[i+1 := u^{+}]) \\
([\ a_1,\ldots,a_n\ ]\ S)[i := u] & = & [\ a_1[i := u],\ldots,a_n[i+(n-1) := u^{n-1}]\ ] \\
 & & \quad S[i+n := u^{+n}] \\
\quad \textbf{where } (\textbf{val}\, t)[i := u] & = & \textbf{val}\, t[i := u] \\
\quad\quad (\textbf{res}\, t\ t')[i := u] & = & \textbf{res}\, t[i := u]\ t'[i := u] \\
(S\ [\ t_1,\ldots t_2\ ])[i := u] & = & S[i := u]\ [\ t_1[i := u],\ldots,t_2[i := u]\ ] \\
(\textbf{rec } [\ a_1,\ldots,a_n\ ]\ S_1 \textbf{ vary } t & = & (\textbf{rec } ([\ a_1,\ldots,a_n\ ]\ S_1)[i := u] \\
\quad \textbf{in } S_2)[i := u] & & \quad \textbf{vary } t[i+n := u^{+n}] \\
 & & \quad \textbf{in } S_2[i+n+2 := u^{+(n+2)}])
\end{array}
$$

Figure 11.3: Substitution in Statements

Free Variables The free variables of statement s are denoted by $\Phi\, s$, satisfying:

$$
\begin{array}{lcl}
\Phi(l_1,\ldots,l_n := t_1,\ldots,t_2) & = & \bigcup_i \Phi\, l_i \cup \bigcup_i \Phi\, t_i \\
\Phi(l_1,\ldots,l_n := \textbf{any } n\ t) & = & \bigcup_i \Phi\, l_i \cup \{\ i-n | i \in \Phi\, t\ \} \\
\Phi(\textbf{var } t_1|t_2 \textbf{ in } S) & = & \Phi\, t_1 \cup \Phi\, t_2 \cup \{\ i-1 | i \in \Phi\, S\ \} \\
\Phi(\textbf{con } t \textbf{ in } S) & = & \Phi\, t \cup \{\ i-1 | i \in \Phi\, S\ \} \\
\Phi\, I & = & \{I\} \\
\Phi(\textbf{proc } S_1 \textbf{ in } S_2) & = & \Phi\, S_1 \cup \{\ i-1 | i \in \Phi\, S_2\ \} \\
\Phi[\ a_1,\ldots,a_n\ ]\ S & = & \bigcup_i \{\ j-(i-1) | j \in \Phi\, a_i\ \} \cup \{\ i-n | i \in \Phi\, S\ \} \\
\quad \textbf{where } \Phi(\textbf{val}\, t) & = & \Phi\, t \\
\quad\quad \Phi(\textbf{res}\, t\ t_i) & = & \Phi\, t \cup \Phi\, t_i \\
\Phi\, S\ [\ t_1,\ldots t_2\ ] & = & \Phi\, S \cup \bigcup_i \Phi\, t_i \\
\Phi(\textbf{rec } [\ a_1,\ldots,a_n\ ]\ S_1 & = & \Phi[\ a_1,\ldots,a_n\ ]\ S_1 \cup \{\ i-n | i \in \Phi\, t\ \} \cup \\
\quad \textbf{vary } t \textbf{ in } S_2) & & \quad \{\ i-n-2 | i \in \Phi\, S_2\ \}
\end{array}
$$

Φ is defined similarly for other statements.

Substitution Substitution of variables arises in the semantics of the statement language. A variable introduced by a (**var** ...) statement can only be substituted by a term, while one introduced by a (**proc** ...) statement can only be substituted by a statement. We define only a single substitution operator: the statement S with the variable at index i replaced by u, a term or statement as appropriate for i, is written $S[i := u]$. When i is zero, this is abbreviated to $S[u]$.

The substitution operator $_[_ := _]$ does not need to rename variables to avoid variable capture, since variables are not used. However, it is necessary to adjust the indices in both S and u whenever the context changes. The statement with free indices incremented (c.f. decremented) by one is written S^+ (c.f. S^-), with an increment (decrement) by n written S^{+n} (S^{-n}). As for terms, indices in a statement can only be decremented if the eliminated indices are not free in the statement. The most important parts of the specification of substitution are shown in Figure 11.3; $_[_ := _]$ is defined similarly for other statements. Parallel substitution is also allowed, with notation $S[i, j := u, v]$.

11.3 Statement Well-formation

In this section, the well-formation rules for simple statements: skip, abort, precondition, sequential composition, if, while, variable and constant declaration and recursion statements are described. The rules cover:

(i) type correctness, and

(ii) restrictions on the use of variables in types.

The well-formation of assignment statements and procedures are covered separately in Section 11.4 and 11.5.

11.3.1 Variables and Constants

The well-formation of a statement (and in particular the terms it contains) depends on the set of program 'variables' which are assignable. Logical constants and, in some circumstances, value parameters are not assignable. An assignable variable environment α is introduced:

$$\alpha \in Assign = \mathbb{N} \nrightarrow \mathbb{B}$$

Constant Terms The predicate $const_term_\alpha : \mathcal{T} \rightarrow \mathbb{B}$ holds for terms which do not contain assignable variables.

$$const_term_\alpha\, t = (\forall i \in \Phi\, t. \neg \alpha\ i)$$

It is convenient to allow value parameters to be used in type declarations, as shown in the example on page 41. Value parameters used in this way are not assignable.

11.3.2 Specification Functions

Three functions are used to specify the well-formation of statements.

Decidable Conditions The predicate $stmt_ok_{\rho,\alpha} : \mathcal{S} \to \mathbb{B}$ specifies the decidable part of the well-formation check.

Parameters The specification $stmt_ok$ uses the function $stmt_fprm_typ_\rho$, which returns the type and mode of the formal parameters of a statement. This function is specified on page 170.

Type Correctness Conditions The well-formation of statements requires the proof of type correctness conditions and disjointness conditions. Consider the type correctness condition for the terms containing the division operator in the following statement:

```
x := 3 ;
IF y /= 0 ==> z := z / y
OR y =  0 ==> z := z / x
END IF
```

In the term `z / y`, `y` is non-zero because of the loop guard `y /= 0`. In the term `z / x`, `x` is non-zero as a result of the initial assignment statement `x := 3`. To formalise this argument, we use the rules for program context, based on the approach described in Section 2.2.6. For statement S in context p the well-formation proof obligation is $stmt_cc_\rho\, p\, S$. The context rule for each statement and the specification of the $fcntx$ function, are in Section 11.10.

The following abbreviations are used:

$$\begin{array}{lcl} stmt_no_fprms_{\rho,\alpha}\, S & \equiv & stmt_ok_{\rho,\alpha}\, S \wedge stmt_fprm_typ_\rho\, S = \langle\rangle \\ bool_term_\rho\, t & \equiv & type_ok_\rho\, t \wedge type_inf_\rho\, t = \mathbb{B} \end{array}$$

11.3.3 Well-formation Rules

The conditions for each form of statement to be well-formed are as follows; the formal specification is shown in Figures 11.4 and 11.5.

Skip, Abort These statements are always well-formed.

Preconditon In the statement (**pre** p **in** S), p must have boolean type. The statement S may not have parameters.

Sequential Composition In $S; T$, the statements S and T may not have parameters.

$$
\begin{array}{lcl}
stmt_ok_{\rho,\alpha}\,\textbf{skip} & = & \mathbf{t} \\
stmt_ok_{\rho,\alpha}\,\textbf{abort} & = & \mathbf{t} \\
stmt_ok_{\rho,\alpha}(\textbf{pre}\ t\ \textbf{in}\ S) & = & bool_term_{\rho}\,t \wedge stmt_no_fprms_{\rho,\alpha}\,S \\
stmt_ok_{\rho,\alpha}\,S_1; S_2 & = & stmt_no_fprms_{\rho,\alpha}\,S_1 \wedge stmt_no_fprms_{\rho,\alpha}\,S_2 \\
stmt_ok_{\rho,\alpha}(\textbf{if}\ \textbf{or}_i\ t_i \Longrightarrow S_i) & = & \bigwedge_{i=1}^{n}(bool_term_{\rho}\,t_i \wedge stmt_no_fprms_{\rho,\alpha}\,S_i) \\
\multicolumn{3}{l}{stmt_ok_{\rho,\alpha}(\textbf{while}\ t_I\ \textbf{vary}\ t_V\ \textbf{loop}\ \textbf{or}_i\ t_i \Longrightarrow S_i)} \\
 & = & bool_term_{\rho}\,t_I \wedge \\
 & & type_ok_{\rho}\,t_V \wedge type_inf_{\rho}\,t_V \approx \mathbb{Z} \wedge \\
 & & \bigwedge_{i=1}^{n}(bool_term_{\rho}\,t_i \wedge stmt_no_fprms_{\rho,\alpha}\,S_i) \\
stmt_ok_{\rho,\alpha}(\textbf{var}\ t|e\ \textbf{in}\ S) & = & type_ok_{\rho}\,t \wedge type_inf_{\rho}\,t = * \wedge const_term_{\alpha}\,t \wedge \\
 & & type_ok_{\rho}\,e \wedge type_inf_{\rho}\,e \approx t \\
 & & stmt_no_fprms_{\rho+t,\alpha+\mathbf{t}}\,S \\
stmt_ok_{\rho,\alpha}(\textbf{con}\ e\ \textbf{in}\ S) & = & type_ok_{\rho}\,e \wedge type_inf_{\rho}(type_inf_{\rho}\,e) = * \wedge \\
 & & stmt_no_fprms_{\rho+type_inf_{\rho}\,e,\alpha+\mathbf{f}}\,S
\end{array}
$$

$$
\begin{array}{l}
stmt_ok_{\rho,\alpha}(\textbf{rec}\ S_1\ \textbf{vary}\ t_v\ \textbf{in}\ S_2)\ = \\
\quad stmt_no_fprms_{\rho',\alpha'}\,S_2 \wedge type_ok_{\rho}\,t_v \wedge type_inf_{\rho}\,t_v \approx \mathbb{Z} \\
\quad \textbf{where}\ \rho' = \rho + type_inf_{\rho}\,t_v + Proc\ (rec_spec\ \langle\rangle\ S_1\ t_v) \\
\quad \phantom{\textbf{where}}\ \ \alpha' = \alpha + \mathbf{f} + \mathbf{f}
\end{array}
$$

Figure 11.4: Specification of Statement Well-formation

If In the statement **if or**$_i$ $t_i \Longrightarrow S_i$, the loop guards t_i must have boolean type. The statements S_i may not have parameters.

Loop In the statement **while** t_I **vary** t_V **loop or**$_i$ $t_i \Longrightarrow S_i$, the invariant t_I and the loop guards t_i must have boolean type. The variant t_V must have integer type. The statements S_i may not have parameters.

Variable Declaration In the statement (**var** $t|e$ **in** S), the term t must be constant (i.e. no free variables which are assignable) and must be of type $*$. The term e must belong to the type t. The statement S may not have parameters. The statement introduces a new variable of type t which may be free in S and which is assignable.

Constant Declaration In the statement (**con** e **in** S), the term e must be of a type which has type $*$. This rule ensures that there is a type t which could be used in a variable declaration: the statement introduces a new constant of type t, which may be free in S. The constant is not assignable. The statement S may not have parameters.

$$
\begin{array}{lll}
\mathit{stmt_cc}_\rho\, p\ \textbf{skip} & = & \mathsf{t} \\
\mathit{stmt_cc}_\rho\, p\ \textbf{abort} & = & \mathsf{t} \\
\mathit{stmt_cc}_\rho\, p\ (\textbf{pre}\ t\ \textbf{in}\ S) & = & (p \Rightarrow \mathit{tcc}_\rho\, t) \wedge \mathit{stmt_cc}_{\rho+t}(p \wedge t)\ S \\
\mathit{stmt_cc}_\rho\, p\ (S_1; S_2) & = & \mathit{stmt_cc}_\rho\, p\ S_1 \wedge \mathit{stmt_cc}_\rho(\mathit{fcntx}_\rho\, p\ S_1)\ S_2 \\
\mathit{stmt_cc}_\rho\, p\ (\textbf{if or}_i\ t_i \Longrightarrow S_i) & = & \bigwedge_{i=1}^{n}((p \Rightarrow \mathit{tcc}_\rho\, t_i) \wedge \mathit{stmt_cc}_\rho(p \wedge t_i)\ S_i) \\
\multicolumn{3}{l}{\mathit{stmt_cc}_\rho\, p\ (\textbf{while}\ t_I\ \textbf{vary}\ t_V\ \textbf{loop or}_i\ t_i \Longrightarrow S_i)} \\
 & = & \mathit{tcc}_\rho\, t_I \wedge \mathit{tcc}_\rho\, t_V \wedge \\
 & & \bigwedge_{i=1}^{n}((t_I \Rightarrow \mathit{tcc}_\rho\, t_i) \wedge \\
 & & \qquad \mathit{stmt_cc}_\rho(t_I \wedge t_i)\ S_i) \\
\mathit{stmt_cc}_\rho\, p\ (\textbf{var}\ t|e\ \textbf{in}\ S) & = & (p \Rightarrow \mathit{tcc}_\rho\, t \wedge \mathit{tcc}_\rho\, e \wedge \mathit{inject_type}_\rho\, e\ t) \wedge \\
 & & (\forall t.\, \mathit{stmt_cc}_{\rho+t}(p^+ \wedge \underline{0} = e^+)\ S) \\
\mathit{stmt_cc}_\rho\, p\ (\textbf{con}\ e\ \textbf{in}\ S) & = & (p \Rightarrow \mathit{tcc}_\rho\, e) \wedge \\
 & & (\forall t \cdot \mathit{stmt_cc}_{\rho+t}(p^+ \wedge \underline{0} = e^+)\ S) \\
 & & \textbf{where} \\
 & & \qquad t = \mathit{type_inf}_\rho\, e \\
\mathit{stmt_cc}_\rho\, p\ (\textbf{rec}\ S_1\ \textbf{vary}\ t_v\ \textbf{in}\ S_2) & = & \mathit{tcc}_\rho\, t_v \wedge (\forall u \cdot (\mathit{stmt_cc}_{\rho'}(\underline{0} = t_v^+)\ S_2)^-) \\
 & & \textbf{where} \\
 & & \qquad u = \mathit{type_inf}_\rho\ t_v \\
 & & \qquad \rho' = \rho + u + \mathit{Proc}\ (\mathit{rec_spec}\ \langle\rangle\ S_1\ t_v)
\end{array}
$$

Figure 11.5: Specification of Statement Correctness Conditions

Recursion In the statement (**rec** S_1 **vary** t **in** S_2), the variant term t must have integer type. The recursion statement introduces a name for the initial value of the variant and a new statement variable; these may be free in S_2. Since S_1 is a component of S_2, there is no need to consider the well-formation of S_1 separately. The function *rec_spec* gives an equivalent statements for the recursive statement variable – see page 190. Parameterised recursion is considered in Section 11.5.

11.4 Well-formation of Assignment Statements

In addition to the issues encountered in Section 11.3, the well-formation rules for assignment statements must include:

(i) absence of aliasing in multiple assignments, and

(ii) correspondance of bound variables with assigned variables in abstract assignment statements.

$$
\begin{array}{lcl}
\mathit{stmt_ok}_{\rho,\alpha}(l_1,\ldots,l_m := t_1,\ldots,t_n) & = & m = n \wedge \mathit{no_alias}\langle l_1,\ldots,l_n\rangle \wedge \\
 & & \bigwedge_{i=1}^{n}(\mathit{left_ok}_{\rho,\alpha}\, l_i \wedge \mathit{type_ok}_{\rho}\, l_i \wedge \\
 & & \quad \mathit{type_ok}_{\rho}\, t_i \wedge \mathit{type_inf}_{\rho}\, t_i \approx \mathit{type_inf}_{\rho}\, l_i) \\
\mathit{stmt_ok}_{\rho,\alpha}(l_1,\ldots,l_m := \mathbf{any}\ n\ t) & = & m = n \wedge \mathit{no_alias}\langle l_1,\ldots,l_n\rangle \wedge \\
 & & \bigwedge_{i=1}^{n}(\mathit{left_ok}_{\rho,\alpha}\, l_i \wedge \mathit{type_ok}_{\rho}\, l_i) \wedge \\
 & & \mathit{bool_term}_{\rho_n}\, t
\end{array}
$$

$$
\textbf{where } \rho_m = \rho + \mathit{type_inf}_{\rho}\, l_1 + (\mathit{type_inf}_{\rho}\, l_2)^{+} + \ldots + (\mathit{type_inf}_{\rho}\, l_m)^{+(m-1)}
$$

$$
\begin{array}{l}
\mathit{stmt_cc}_{\rho}\, p\ (l_1,\ldots,l_m := t_1,\ldots,t_n) \ = \ p \Rightarrow \\
\quad \bigwedge_{(i,t)\in \mathit{asgn}\ \langle l_1,\ldots,l_n\rangle\ \langle t_1,\ldots,t_n\rangle}(\mathit{inject_type}_{\rho}\, t\ (\mathit{type_inf}_{\rho}\, \underline{i})) \wedge \\
\quad \bigwedge_{i=1}^{n}(\mathit{tcc}_{\rho}\, l_i \wedge \mathit{tcc}_{\rho}\, t_i) \wedge \mathit{no_alias_cc}\langle l_1,\ldots,l_n\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{stmt_cc}_{\rho}\, p\ (l_1,\ldots,l_m := \mathbf{any}\ n\ q) \ = \ p \Rightarrow \\
\quad \bigwedge_{i=1}^{m}(\mathit{tcc}_{\rho}\, l_i) \wedge \mathit{no_alias_cc}\langle l_1,\ldots,l_m\rangle \wedge \\
\quad (\forall t_1 \cdot (\forall \cdot t_2 \ldots \cdot (\forall t_m \cdot \mathit{tcc}_{\rho_m}\, q \wedge \\
\qquad (q \Rightarrow \bigwedge_{(i,t)\in \mathit{asgn}\ \langle l_1^{+n},\ldots,l_n^{+n}\rangle\ \langle \underline{n-1},\ldots,\underline{0}\rangle}(\mathit{inject_type}_{\rho_m}\, t\ (\mathit{type_inf}_{\rho_m}\, \underline{i}))) \ldots))) \\
\qquad \textbf{where } t_i \ = \ (\mathit{type_inf}_{\rho}\, l_i)^{+(i-1)} \\
\qquad\qquad\quad \rho_m \ = \ \rho + t_1 + \ldots + t_m
\end{array}
$$

Figure 11.6: Assignment Statement Well-formation and Correctness

---

The well-formation rules and correctness conditions[5] for the two forms of assignments statement are shown in Figure 11.6.

Multiple Assignment In $l_1, \ldots, l_m := t_1, \ldots, t_n$, the left values l_i must be well-formed as terms and as left values. The left values must not alias. The terms t_i must correspond to the left values in type and number.

Abstract Assignment In $l_1, \ldots, l_m :=$ **any** $n\ t$, the left values l_i must be well-formed as terms and as left values. The left values must not alias. The integer n, the number of bound variables in the concrete systax, must be equal to the number of left values m. The term t must be of boolean type: there are n additional variables free in t of the type of the corresponding left value.

[5]The well-formation specification uses the $\approx$ relation for (decidable) type compatibility; in fact, unification is used to infer any implicit arguments occuring on either the left or right hand sides of the assignments and similarly in parameters. The Haskell versions of the specifications for the multiple assignment are shown in Figures 5.2 and 5.3.

Type Correctness The type rule for an assignment statement $l := t$ is the same as the type rule for the term $l = t$:

$$inject_type_\rho\, t\ (type_inf_\rho\, l)$$

However, the correspondence is complicated when multiple (but non-overlapping) elements of a variable are assigned in a multiple assignment and a subtype constraints requires a relationship between the elements of the variable to be maintained. The difficulty is illustrated by the following example. Let p be the predicate $(\lambda a : Array\ 10\ \mathbb{N} \cdot (\forall i : upto\ 5 \cdot a\ (i + 5) = 2 * (a\ i)))$ and let $a : \{p\}$ be a variable. then the assignment $a\ 0, a\ 5 := m, n$ is well-typed only if $n = 2 * m$. However. this correctness condition would not be required to type check the predicate $a\ 0 = m \wedge a\ 5 = n$. To get the right correctness condition, the equivalent assignment $a := a \oplus (0, m) \oplus (5, n)$ should be used, where multiple assignments to the same variable have been gathered together. The *asgn* function is used to transform multiple assignments; it is described on page 182.

11.4.1 Well-formation and Aliasing of Left Values

To be well-formed, a left value – a term on the left-hand side of an assignment – must have an assignable variable at its head and any 'accessor functions' used in a left value must be an accessor. These well-formation rules are specified by the function $left_ok_{\rho,\alpha} : \mathcal{L} \to \mathbb{B}$.

$$\begin{array}{lcl} left_ok_{\rho,\alpha}\, \underline{i} & = & i \in \alpha \\ left_ok_{\rho,\alpha}\, f\ a & = & left_ok_{\rho,\alpha}\, f \\ left_ok_{\rho,\alpha}\, h.a\ l & = & left_ok_{\rho,\alpha}\, l \wedge (\exists d : \mathcal{I}; c : \mathcal{I} \cdot \\ & & \quad is_adt_\rho\, h.d \wedge h.c \in adt_cons_ids_\rho\, h.d \wedge \\ & & \quad h.a \in adt_acc_ids_\rho\, h.d\ h.c) \\ left_ok_{\rho,\alpha}\, l\,\hat{}\,t & = & left_ok_{\rho,\alpha}\, l \end{array}$$

This function is used in Figure 11.6 and also in the well-formation specification for actual parameters on page 173. Note that a left value is also required to be a well-formed term.

Assigned Variable and Selectors To specify the checks for aliasing, two 'syntactic' functions are introduced:

1. The function $lhs : \mathcal{L}_i \to \mathbb{N}$ returns the variable which is assigned in a left value.

$$\begin{array}{lcl} lhs\, \underline{i} & = & i \\ lhs(f\ a) & = & lhs\, f \\ lhs(h.a\ l) & = & lhs\, l \\ lhs(l\,\hat{}\,t) & = & lhs\, l \end{array}$$

2. The function $left_selector : \mathcal{L}_i \to \textbf{seq}\,\mathcal{E}$ converts a left value to a list of selectors ($\mathcal{E}$), where:

$$\mathcal{E} ::= \textbf{app}\,\mathcal{T} \mid \textbf{map}\,\mathcal{T} \mid \textbf{acc}\,h.a$$

The function is specified by:

$$\begin{array}{lcl} left_selector\ \underline{i} & = & \langle\rangle \\ left_selector(f\ a) & = & left_selector\ f \frown \langle \textbf{app}\ a\rangle \\ left_selector(l\ \hat{}\ t) & = & left_selector\ l \frown \langle \textbf{map}\ t\rangle \\ left_selector(h.a\ l) & = & left_selector\ l \frown \langle \textbf{acc}\ h.a\rangle \end{array}$$

These functions can be used to characterise a left value as a list of selectors applied to a variable. For example, using the example type and variable declarations introduced on page 155:

Left Value	Variable	Selectors
$one\ (ad\ t)$	ad	$\langle \textbf{app}\ t, \textbf{acc}\ one\rangle$
$three\ (ad\ t)\ u\ \hat{}\ v$	ad	$\langle \textbf{app}\ t, \textbf{acc}\ three, \textbf{app}\ u, \textbf{map}\ v\rangle$

Note that the list starts with the innermost selector.

11.4.2 Aliasing of Left Values

The variables used on the left hand side of an assignment must not alias. A multiple assignment statement may contain more than one assignment to the same variable, provided that the components of the variable selected are disjoint. Disjointness of assignments to functions and maps is not necessarily decidable. As a result, the check for the absence of aliasing has two parts:

(i) a decidable check, and

(ii) a 'disjointness condition' proof obligation.

Simple Variables Let x, y be variables. Consider the following pairs of left values:

x	x	aliasing of x
x	y	no aliasing, since x and y are distinct variables

Functions and Maps Let f, g be variables of function type and a, b be terms. Consider the following pairs of left values:

$f\ a$	$g\ a$	no aliasing, since f, g are distinct variables
$f\ a$	$f\ b$	possible aliasing unless $a \neq b$; this condition must be proved
$f\ a\ a'$	$f\ b\ b'$	possible aliasing unless $a \neq b$ or $a' \neq b'$
$f\ a\ a'$	$f\ b\ b'\ b''$	possible aliasing unless $a \neq b$ or $a' \neq b'$; the value of b'' does not effect aliasing

Datatypes Let d be a variables of datatype D with accessors *one* and *two*. Consider the following pairs of left values:

one d	*one* d	aliasing, since same accessor used
one d	*two* d	no aliasing

The disjointness of left values for datatype variables are always decidable – the accessors are either the same or different. However, datatype accessors may be used together with function or map. For example:

Left Values		Disjointness Condition
$one\ (ad\ t)$	$one\ (ad\ t')$	$t \neq t'$
$three\ (ad\ t)\ u \,\hat{}\, v$	$three\ (ad\ t')\ u' \,\hat{}\, v'$	$t \neq t' \vee u \neq u' \vee v \neq v'$

Decidable No-Aliasing Checks The decidable checkes are specified by the function $no_alias : \mathbf{seq}\,\mathcal{L} \to \mathbb{B}$.

$$
\begin{aligned}
no_alias\langle\rangle &= \mathbf{t} \\
no_alias(\langle l\rangle \frown ls) &= sels_no_alias(map\ \ left_selector(\langle l\rangle \frown ss)) \wedge no_alias\ ds \\
&\quad \mathbf{where}\ (ss, ds) = same_assigned(lhs\ ls)\ ls
\end{aligned}
$$

The function *same_assigned* separates the list of left values ls into those (ss) with the same assigned variable as l, and those with a different one (ds). The specification of this function is omitted.

The predicate $sels_no_alias : \mathbf{seq}(\mathbf{seq}\,\mathcal{E}) \to \mathbb{B}$ checks for aliasing between the selector sequences of a particular variable. A single selector sequence cannot alias; sequences of two or more must be pairwise free of aliasing, as shown by the predicate $sel_no_alias : \mathbf{seq}\,\mathcal{E} \to \mathbf{seq}\,\mathcal{E} \to \mathbb{B}$. This predicate is true provided it is *possible* that the selector lists do not alias, using the following rules:

(i) an empty list of selectors aliases,

(ii) selection of a function or map element ensures the possibility of no aliasing,

(iii) selecting using datatype accessors ensures that there is no aliasing if the accessors are different or if the rest of the selectors do not alias.

Note that the type-correctness of the left values ensures that the corresponding selectors in two lists of selectors for the same variable are of the same sort – function, map or datatype accessor – as least up to corresponding selectors with different accessors. These functions are specified by:

$$
\begin{array}{lcl}
sels_no_alias\langle ss\rangle & = & \mathbf{t} \\
sels_no_alias\langle ss\rangle \frown sss & = & \bigwedge_{ss' \in sss} sel_no_alias\; ss\; ss' \wedge sels_no_alias\; sss
\end{array}
$$

$$
\begin{array}{lcl}
sel_no_alias\langle\rangle\; ss & = & \mathbf{f} \\
sel_no_alias\; ss\; \langle\rangle & = & \mathbf{f} \\
sel_no_alias(\langle \mathbf{app}\, t\rangle \frown ss)\; (\langle \mathbf{app}\, t'\rangle \frown ss') & = & \mathbf{t} \\
sel_no_alias(\langle \mathbf{map}\, t\rangle \frown ss)\; (\langle \mathbf{map}\, t'\rangle \frown ss') & = & \mathbf{t} \\
sel_no_alias(\langle \mathbf{acc}\, h.a\rangle \frown ss)\; (\langle \mathbf{acc}\, h.a\rangle \frown ss') & = & sel_no_alias\; ss\; ss' \\
sel_no_alias(\langle \mathbf{acc}\, h.a\rangle \frown ss)\; (\langle \mathbf{acc}\, h.a'\rangle \frown ss') & = & \mathbf{t}
\end{array}
$$

Disjointness Conditions The 'disjointness condition' proof obligation is specified by the function $no_alias_cc : \mathbf{seq}\,\mathcal{L} \rightarrow \mathcal{T}$. The function $sels_no_alias_cc : \mathbf{seq}(\mathbf{seq}\,\mathcal{E}) \rightarrow \mathcal{T}$ gives the condition for the absence of aliasing in the selector sequences for a particular variable. A single selector sequence cannot alias; two or more sequences must be pairwise free of aliasing, as shown by the function $sel_no_alias_cc : \mathbf{seq}\,\mathcal{E} \rightarrow \mathbf{seq}\,\mathcal{E} \rightarrow \mathcal{T}$. These functions are specified by:

$$
\begin{array}{lcl}
no_alias_cc\; \langle\rangle & = & \mathbf{t} \\
no_alias_cc(\langle l\rangle \frown ls) & = & sels_no_alias_cc(map\;\; left_selector(\langle l\rangle \frown ss)) \wedge no_alias\; ds \\
 & & \mathbf{where}\; (ss, ds) = same_assigned(lhs\; ls)\; ls
\end{array}
$$

$$
\begin{array}{lcl}
sels_no_alias_cc\langle ss\rangle & = & \mathbf{t} \\
sels_no_alias_cc\langle ss\rangle \frown sss & = & \bigwedge_{ss' \in sss} sel_no_alias_cc\; ss\; ss' \wedge sels_no_alias_cc\; sss
\end{array}
$$

$$
\begin{array}{lcl}
sel_no_alias_cc\langle\rangle\; ss & = & \mathbf{f} \\
sel_no_alias_cc\; ss\; \langle\rangle & = & \mathbf{f} \\
sel_no_alias_cc(\langle \mathbf{app}\, t\rangle \frown ss)\; (\langle \mathbf{app}\, t'\rangle \frown ss') & = & t \neq t' \vee sel_no_alias_cc\; ss\; ss' \\
sel_no_alias_cc(\langle \mathbf{map}\, t\rangle \frown ss)\; (\langle \mathbf{map}\, t'\rangle \frown ss') & = & t \neq t' \vee sel_no_alias_cc\; ss\; ss' \\
sel_no_alias_cc(\langle \mathbf{acc}\, h.a\rangle \frown ss)\; (\langle \mathbf{acc}\, h.a\rangle \frown ss') & = & sel_no_alias_cc\; ss\; ss' \\
sel_no_alias_cc(\langle \mathbf{acc}\, h.a\rangle \frown ss)\; (\langle \mathbf{acc}\, h.a'\rangle \frown ss') & = & \mathbf{t}
\end{array}
$$

Using these rules, the disjointness condition for two left values that never alias may be more complex than $\mathbf{t}$; for example, the left values $one\ (ad\ t)$ and $two\ (ad\ u)$ have the condition $t \neq u \vee \mathbf{t}$. This function is used in the specification of correctness conditions for assignment statements, shown in Figure 11.6 and to specify the condition for the absence of aliasing in actual parameter lists in Figure 11.10.

11.5 Procedure Statements Well-formation

The well-formation rules for 'procedure' statements, including call, abstraction and application are described in this section. In addition to the forms of rule already

$$
\begin{array}{lcl}
stmt_ok_{\rho,\alpha}\, \underline{I} & = & \mathbf{t}\ \mathbf{if}\ \rho\ !!\ I\ \textit{is a statement} \\
stmt_ok_{\rho,\alpha}(\mathbf{proc}\ S_1\ \mathbf{in}\ S_2) & = & stmt_ok_{\rho,\alpha}\, S_1 \wedge stmt_no_fprms_{\rho',\alpha+\mathbf{f}}\, S_2 \\
 & & \mathbf{where}\ \rho' = \rho + Proc\ S_1 \\
stmt_ok_{\rho,\alpha}[\, a_1, \ldots, a_n\,]\, S & = & fprm_ok^{\rho',\alpha'}_{\rho,\alpha} \langle a_1, \ldots, a_n \rangle \wedge stmt_no_fprms_{\rho',\alpha'}\, S \\
stmt_ok_{\rho,\alpha}\, S\, [\, t_1, \ldots t_n\,] & = &
\end{array}
$$

$$
\begin{array}{l}
\quad stmt_ok_{\rho,\alpha}\, S \wedge \bigwedge_{i=1}^{n} (type_ok_{\rho}\, t_i) \wedge \\
\quad prm_compat_{\rho,\alpha}(stmt_indx\ S)\ (stmt_fprm_typ_{\rho}\, S)\ \langle t_1, \ldots, t_n \rangle \\
stmt_ok_{\rho,\alpha}(\mathbf{rec}\ [\ as\ ]\ S_1\ \mathbf{vary}\ t\ \mathbf{in}\ S_2) = \\
\quad fprm_ok^{\rho',\alpha'}_{\rho,\alpha}\, as \wedge type_ok_{\rho'}\, t \wedge type_inf_{\rho'}\, t \approx \mathbb{Z} \wedge stmt_no_fprms_{\rho'',\alpha''}\, S_2 \\
\quad \mathbf{where}\ \rho'' \ = \ \rho' + type_inf_{\rho}\, t + Proc\ (rec_spec\ as\ S_1\ t) \\
\qquad\qquad \alpha'' \ = \ \alpha' + \mathbf{f} + \mathbf{f}
\end{array}
$$

Figure 11.7: Specification of Procedure Statements Well-formation

encountered, the rules for procedures include:

(i) absence of aliasing in application statements,

(ii) compatibility between the actual and formal parameters in an application statements.

11.5.1 Well-formation Rules

Call The procedure call $\underline{I}$ is well-formed if I is a statement variable[6].

Procedure Block In **proc** S_1 **in** S_2, the statements S_1 and S_2 must be well-formed. A new statement variable, which may be free in S_2, is added to the environment.

Application In the application $S\ [\ t_1, \ldots t_n\ ]$, the statement S must be well-formed, with n formal parameters. The actual parameters t_i must be well-formed and compatible with the formal parameters, without aliasing. The rules for compatibility and the absence of aliasing are described in more detail below.

The well-formation rules for procedure and parameterisation statements are specified in Figure 11.7; correctness conditions are in Figure 11.8. Cases are also shown for parameterised recursion. These definitions make use of functions which are defined below:

[6]There should be a corresponding check in the specification for a ordinary variable (see Section 9.4.3), to prevent statement variables appearing in terms. For implementation, it is easy to distinguish statement variables from other variables during encoding.

Function	Description	Page
stmt_fprm_typ	Types and modes of the formal parameters of a statement	170
fprm_ok	Decidable well-formation checks for formal parameters	171
fprm_cc	Proof obligations for formal parameter well-formation	171
prm_compat	Decidable checks on compatibility of formal and actual parameters	172
prm_compat_cc	Proof obligations for compatibility of formal and actual parameters	172
stmt_indx	Statement index	176
rec_spec	Recursion specification	190

11.5.2 Parameterised Statements

The function $stmt_fprm_typ_\rho : \mathcal{S} \rightarrow \mathbf{seq}(\mathcal{T} \times \mathcal{M})$ returns the formal parameters of a statement. Formal parameters are introduced by an abstraction statement: $[a_1, a_2]S$.

$$\mathcal{M} ::= \mathbf{val} \mid \mathbf{valres} \mid \mathbf{res} \mid \mathbf{ref}$$

$$\begin{array}{rcl}
stmt_fprm_typ_\rho\, I & = & stmt_fprm_typ_\rho\ S \\
 & & \mathbf{where}\ \rho\ !!\ \ I\ is\ a\ statement \\
stmt_fprm_typ_\rho[\ a_1, \ldots, a_n\ ]\ S & = & \langle type_mode\, a_1, \ldots, type_mode\, a_n \rangle \\
\mathbf{where}\ type_mode\, \mathbf{val}\, t & = & t, \mathbf{val} \\
type_mode\, \mathbf{valres}\, t & = & t, \mathbf{valres} \\
type_mode\, \mathbf{res}\, t\ t_i & = & t, \mathbf{res} \\
type_mode\ \mathbf{ref}\ t & = & t, \mathbf{ref} \\
stmt_fprm_typ_\rho(\mathbf{rec}\ S_1\ \mathbf{vary}\ t\ \mathbf{in}\ S_2) & = & stmt_fprm_typ_\rho\, S_1 \\
stmt_fprm_typ_\rho\, S & = & \langle\rangle\ \mathbf{otherwise}
\end{array}$$

The specification of well-formation (function $stmt_ok_{\rho,\alpha}$) allows a statement with parameters to be used only as the specifying statement of a procedure declaration or recursion block. With these restrictions[7], the possible forms of statement in an application (i.e. for S in $S[x_1, x_2]$) are the statement variables introduced by a procedure declaration or recursion block, or the recursion block itself.

11.5.3 Well-formation of Formal Parameters

The formal parameters using in an abstraction $[\ a_1, \ldots, a_n\ ]\ S$, must have types which are constant (i.e. no free variables which are assignable) and of type $*$.

[7]The restrictions on the use of parameters are enforced by the concrete syntax given in Appendix A.

$$
\begin{array}{lcl}
\mathit{stmt_cc}_\rho\, p\ I & = & \mathbf{t} \\
\mathit{stmt_cc}_\rho\, p\ (\mathbf{proc}\ S_1\ \mathbf{in}\ S_2) & = & \mathit{stmt_cc}_\rho\, \mathbf{t}\ S_1 \wedge (\mathit{stmt_cc}_{\rho'}\, p^+\ S_2)^- \\
 & & \mathbf{where}\ \rho' = \rho + \mathit{Proc}\ S_1 \\
\mathit{stmt_cc}_\rho\, p\ [\ a_1, \ldots, a_n\ ]\ S & = & \mathit{fprm_cc}_\rho\, p\ \langle a_1, \ldots, a_n \rangle\ (\lambda \rho' \cdot \lambda p' \cdot \mathit{stmt_cc}_{\rho'}\, p'\ S) \\
\mathit{stmt_cc}_\rho\, p\ (S\ [\ t_1, \ldots, t_n\ ]) & = & \mathit{stmt_cc}_\rho\, p\ S \wedge (p \Rightarrow \bigwedge_{i=1}^{n} (\mathit{tcc}_\rho\, t_i) \wedge \\
 & & \mathit{prm_compat_cc}_\rho(\mathit{stmt_fprm_typ}_\rho\, S)\ \langle t_1, \ldots, t_n \rangle) \\
\mathit{stmt_cc}_\rho\, p\ (\mathbf{rec}\ [\ as\ ]\ S_1 & = & \mathit{fprm_cc}_\rho\, \mathbf{t}\ as\ (\lambda \rho' \cdot \lambda p' \cdot \\
\quad \mathbf{vary}\ t\ \mathbf{in}\ S_2) & & \quad (p' \Rightarrow \mathit{tcc}_{\rho'}\, t) \wedge (\forall u \cdot (\mathit{stmt_cc}_{\rho''}\, p'^{+2}\ S_2)^-)) \\
 & & \mathbf{where} \\
 & & \quad u = \mathit{type_inf}_{\rho'}\, t \\
 & & \quad \rho'' = \rho' + u + \mathit{Proc}\ (\mathit{rec_spec}\, as\ S_1\ t)
\end{array}
$$

Figure 11.8: Specification of Procedure Statement Correctness Conditions

$$
\begin{array}{lcl}
\mathit{fprm_ok}^{\rho;\alpha}_{\rho;\alpha}\langle\rangle & = & \mathbf{t} \\
\mathit{fprm_ok}^{\rho'',\alpha''}_{\rho,\alpha}\langle \mathbf{val}\, t \rangle \frown as & = & \mathit{type_ok}_\rho\, t \wedge \mathit{type_inf}_\rho\, t = * \wedge \\
 & & \mathit{const_term}_\alpha\, t \wedge \mathit{fprm_ok}^{\rho'',\alpha''}_{\rho+t,\alpha'}\, as \\
\quad \mathbf{where}\ \alpha' = \alpha + \mathbf{f}\ \mathbf{if}\ \underline{0}\ \text{is free in}\ as & & \\
\quad \phantom{\mathbf{where}\ }\alpha' = \alpha + \mathbf{t}\ \mathbf{otherwise} & & \\
\mathit{fprm_ok}^{\rho'',\alpha''}_{\rho,\alpha}\langle \mathbf{res}\, t\ t_i \rangle \frown as & = & \mathit{type_ok}_\rho\, t \wedge \mathit{type_inf}_\rho\, t = * \wedge \\
 & & \mathit{type_ok}_\rho\, t_i \wedge \mathit{type_inf}_\rho\, t \approx t \wedge \\
 & & \mathit{const_term}_\alpha\, t \wedge \mathit{fprm_ok}^{\rho'',\alpha''}_{\rho+t,\alpha+\mathbf{t}}\, as \\
\mathit{fprm_ok}^{\rho'',\alpha''}_{\rho,\alpha}\langle \mathbf{m}\ t \rangle \frown as & = & \mathit{type_ok}_\rho\, t \wedge \mathit{type_inf}_\rho\, t = * \wedge \\
 & & \mathit{const_term}_\alpha\, t \wedge \mathit{fprm_ok}^{\rho'',\alpha''}_{\rho+t,\alpha+\mathbf{t}}\, as \\
 & & \mathbf{where}\ \mathbf{m}\ \text{is}\ \mathbf{valres}\ \text{or}\ \mathbf{ref} \\
 & & \\
\mathit{fprm_cc}_\rho\, p\ \langle\rangle\ F & = & F\ \rho\ p \\
\mathit{fprm_cc}_\rho\, p\ (\langle \mathbf{m}\ t \rangle \frown as)\ F & = & (p \Rightarrow \mathit{tcc}_\rho\, t) \wedge (\forall t \cdot \mathit{fprm_cc}_{\rho+t}\, p^+\ as\ F^+) \\
 & & \mathbf{where}\ \mathbf{m}\ \text{is}\ \mathbf{val},\ \mathbf{valres}\ \text{or}\ \mathbf{ref} \\
\mathit{fprm_cc}_\rho\, p\ (\langle \mathbf{res}\, t\ t_i \rangle \frown as)\ F & = & (p \Rightarrow \mathit{tcc}_\rho\, t \wedge \mathit{tcc}_\rho\, t_i \wedge \mathit{inject_type}_\rho\, t_i\ t) \wedge \\
 & & (\forall t \cdot \mathit{fprm_cc}_{\rho+t}(p^+ \wedge \underline{0} = t_i^+)\ as\ F^+)
\end{array}
$$

Figure 11.9: Formal Parameters Well-Formation and Correctness

$$
\begin{aligned}
&prm_compat_{\rho,\alpha}\ I\ \langle(f_0, m_0)\ldots,(f_I, m_I)\rangle\ \langle a_0,\ldots,a_J\rangle = (I = J) \wedge \\
&\quad \textstyle\bigwedge_{i\in 0\ldots I}(p_type_ok_{\rho_i,\alpha_i}\ m_i\ (f_i[i-1,\ldots,0 := a_0^{+i},\ldots,a_{i-1}^{+i}])^{-i}\ a_i^{+(i-1)}) \wedge \\
&\quad p_no_alias_{\alpha}\ I\ \langle m_0,\ldots,m_I\rangle\ \langle a_0,\ldots,a_J\rangle
\end{aligned}
$$

$$
\begin{aligned}
&prm_compat_cc_{\rho}\langle(f_0, m_0)\ldots,(f_n, m_n)\rangle\ \langle a_0,\ldots,a_n\rangle = \\
&\quad inject_type_{\rho}\, a_0\ f_0 \wedge \\
&\quad inject_type_{\rho}\, a_1\ (f_1[0 := a_0^{+}])^{-} \wedge \\
&\qquad \vdots \\
&\quad inject_type_{\rho}\, a_n\ (f_n[n-1,\ldots,0 := a_0^{+n},\ldots,a_{n-1}^{+n}])^{-n} \wedge \\
&\quad no_alias_cc(out_param\langle m_0,\ldots,m_n\rangle\ \langle a_0,\ldots,a_n\rangle)
\end{aligned}
$$

Figure 11.10: Parameter Compatibility

Each parameter declaration introduces a new variable which may be free in the statement S. A parameter of **val** mode may also be free in subsequent parameter types, as described on page 41. The new variable introduced by a parameter declaration is assignable unless the type of a subsequent parameter is dependent on the variable. A formal parameter of mode **res** has an initial value which must be assignment compatible with the parameter type.

The decidable well-formation check is described by the function $fprm_ok_{\rho,\alpha}^{\rho',\alpha}$: $\mathbf{seq}\,\mathcal{A}\rightarrow\mathbb{B}$ where the initial environment is ρ, α and the environment with formal parameters added is ρ', α'. Proof obligations for the well-formation of formal parameters are described by the function $fprm_cc_{\rho} : \mathcal{T} \rightarrow \mathbf{seq}\,\mathcal{A} \rightarrow (TypEnv \rightarrow \mathcal{T} \rightarrow \mathcal{T}) \rightarrow \mathcal{T}$. The arguments of this function are the context, the list of formal parameters and, rather than the parameterised statement, a function to which the environment and context created by the parameters are applied to give the type correctness conditions. The specifications are in Figure 11.9.

11.5.4 Compatibility of Actual and Formal Parameters

For actual and formal parameters to be compatible:

(i) there must be equal numbers of formal and actual parameters: correspondence between formal and actual is by position,

(ii) the actual parameter must belong to the type declared for the formal parameter

(iii) the actual parameters corresponding to result, value-result or reference formal parameter modes must be left-values, and

(iv) the actual parameters must not give rise to aliasing.

As for the assignment statement, type compatibility and aliasing rules require proof obligations.

Decidable Compatibility Rules The decidable rules for type compatibility and absence of aliasing between formal and actual parameter lists are described by the function $prm_compat_{\rho,\alpha} : \mathbb{N} \rightarrow \mathbf{seq}(\mathcal{T} \times \mathcal{M}) \rightarrow \mathbf{seq}\,\mathcal{T} \rightarrow \mathbb{B}$. The specification is in Figure 11.10. Each parameter is added to the environment of the subsequent parameters, since the type of a parameter may depend on a preceding parameter. The variable introduced by a parameter declaration is assignable unless the parameter is of **val** mode and the variable is free in the type of a following parameter.

The function $p_type_ok_{\rho,\alpha} : \mathcal{M} \rightarrow \mathcal{T} \rightarrow \mathcal{T} \rightarrow \mathbb{B}$ specifies the type check for a parameter of each mode and the requirement for parameters of modes other than **val** to be a well-formed left value.

$$\begin{array}{lcl} p_type_ok_{\rho,\alpha}\,\mathbf{val}\ f\ a & = & type_inf_{\rho}\,a \approx f \\ p_type_ok_{\rho,\alpha}\,\mathbf{m}\ f\ a & = & left_ok_{\rho,\alpha}\,a \wedge type_inf_{\rho}\,a \approx f \\ & & \mathbf{where}\ \mathbf{m}\ \text{is}\ \mathbf{valres},\ \mathbf{res}\ \text{or}\ \mathbf{ref} \end{array}$$

Proof Obligations for Parameter Compatibility Parameter compatibility requires proof obligation for:

(i) type correctness conditions for the actual parameters, and

(ii) the absence of aliasing between result and reference parameters in an actual parameter list.

The type correctness condition ensures that each actual parameter belongs to the type given in the formal parameter declaration, with the preceeding actual parameter values substituted where the formal parameter type is dependent on an earlier formal parameter. The anti-aliasing 'disjointness condition' is the same as that for an assignment statement, ignoring **val** parameters: the function *no_alias_cc* is defined on page 168. The function *prm_compat_cc* is used in the specification of correctness conditions (Figure 11.8) for the application statement. The function $prm_compat_cc_{\rho} : \mathbf{seq}(\mathcal{T} \times \mathcal{M}) \rightarrow \mathbf{seq}\,\mathcal{T} \rightarrow \mathcal{T}$ describes these proof obligations. The specification is in Figure 11.10.

The function $out_param : \mathbf{seq}\,\mathcal{M} \rightarrow \mathbf{seq}\,\mathcal{T} \rightarrow \mathbf{seq}\,\mathcal{L}$ selects actual parameters which correspond to formal result or reference parameters:

$$
\begin{array}{lcl}
out_param\langle\rangle\ \langle\rangle & = & \langle\rangle \\
out_param(\langle\mathbf{val}\rangle \frown ms)\ (\langle a\rangle \frown as) & = & out_param\ ms\ as \\
out_param(\langle m\rangle \frown ms)\ (\langle a\rangle \frown as) & = & \langle a\rangle \frown out_param\ ms\ as \\
 & & \textbf{where } m \text{ is } \textbf{valres}, \textbf{res} \text{ or } \textbf{ref}
\end{array}
$$

11.5.5 Aliasing of Parameters

Two forms of aliasing can occur in an actual parameter list

(i) aliasing between result (**res**, **valres**) or reference parameter which are not disjoint, and

(ii) aliasing between **ref** parameters and global variables.

The two cases are considered in more detail below and the rules for avoiding aliasing given.

Aliasing of Result and Reference Parameters The following example illustrates this form of aliasing: is z increased by 1 or 2?

```
PROC Inc IS [VALRES x:Nat; VALRES y:Nat]
  x,y := x+1,y+2
IN
  Inc[z, z]
END PROC
```

This form of aliasing is equivalent to aliasing in assignment statements. In the example, the equivalent assignment is `z,z := z+1,z+2`. As with assignment, both a static check and a disjointness proof obligation may be required. Parameters of mode **val** cannot give rise to aliasing.

Aliasing Between Globals and Reference Parameters The following example illustrates this form of aliasing: is G doubled or not?

```
VAR G : Nat IN
  PROC Inc IS [REF x:Nat]
    x,G := x + G, x
  IN
    Inc[G]
```

```
    END PROC
  END VAR
```

To prevent aliasing between a global variable and a reference parameter, we must ensure that the variable used as the actual parameter cannot be free in the statement which gives the meaning of the statement variable. Thus, if $X = [\ \mathbf{ref}\ r : t\ ]\ S$ then $\underline{X}\ [\ g\ ]$ does not cause aliasing provided that $g \notin \Phi\, S$. Monotonicity of call by reference is proved by Back [Bac87] with this side-condition[8].

It is difficult to implement Back's side-condition directly, since the set of free variables is not preserved by refinement. As a result, to check the well-formation of a call to the procedure, the implementation of a procedure would need to be examined; we would prefer to have the well-formation of the call depend only on the specifying statement of a procedure, not its refinement. We adopt the following rule which is more restrictive than Back's.

Rule to Prevent Aliasing with Globals If S is a statement with formal parameter $\mathbf{ref}\ r : t$ then the application $S\ [\ g\ ]$ does not alias with global variables provided that the variable g was declared *after* the formal parameters of S and therefore g cannot be free in S.

At a practical level, a programmer wishing to update an object from within a procedure body without using a result parameter has two options

(i) declare the object first and refer to it from the procedure body as a global variable, or

(ii) declare the procedure first and refer to the object using a reference parameter.

With this rule, recursion blocks cannot have reference parameters[9].

Decidable Anti-Aliasing Checks The function $p_no_alias_\alpha$ specifies the decidable anti-alising checks. To prevent aliasing between result, value result and reference parameters, the variables used must be disjoint in the same way that variables in a multiple assignment must be disjoint, as specified by the function no_alias (see page 167).

$$p_no_alias_\alpha\ I\ ms\ as = no_alias(out_param\ ms\ as) \wedge no_alias_glb_\alpha\ I\ ms\ as$$

[8] The use of copy-in copy-out semantics for **valres** and **res** parameters ensures that there is no aliasing in the second example if **ref** is replaced by **valres**. With this change, `Inc[G]` is well-formed and equivalent to `G := G + G`. Although this is a well-formed statement, it is *confusing* and it is unlikely to reflect the true intentions of a programmer.

[9] However, a recursive procedure can have reference parameters.

The rule to prevent aliasing with globals can be implemented by comparing the indices of the statement variable and the actual parameter variable. If the index X is defined as [**ref** $r : t$] S then the call $\underline{X}$ [$\underline{g}$] is well-formed if $g < X$.

This implementation is used in the function $no_alias_glb_\alpha : \mathbb{N} \to \mathbf{seq}\,\mathcal{M} \to \mathbf{seq}\,\mathcal{T} \to \mathbb{B}$.

$$
\begin{array}{lcl}
no_alias_glb_\alpha\ I\ \langle\rangle\ \langle\rangle & = & \mathsf{t} \\
no_alias_glb_\alpha\ I\ (\langle \mathbf{ref} \rangle \frown ms)\ (\langle a \rangle \frown as) & = & lhs\ a < I \wedge no_alias_glb_\alpha\ I\ ms\ as \\
no_alias_glb_\alpha\ I\ (\langle m \rangle \frown ms)\ (\langle a \rangle \frown as) & = & no_alias_glb_\alpha\ I\ ms\ as \\
 & & \quad \textbf{for other parameter modes}
\end{array}
$$

The index of the statement variable is returned by the function $stmt_indx : \mathcal{S}_i \to \mathbb{N}$, which is used in the definition of $stmt_ok$ in Figure 11.7, on page 169.

$$
\begin{array}{lcl}
stmt_indx\ \underline{I} & = & I \\
stmt_indx(\mathbf{rec}\ S_1\ \mathbf{vary}\ t\ \mathbf{in}\ S_2) & = & 0
\end{array}
$$

Anti-Aliasing Proof Obligations Proof obligations are only required to check the absence of aliasing between result and reference parameters and these checks are the same as those required for assignment statements. The function *no_alias_cc*, defined on page 168 is used in the definition of *prm_compat_cc* above.

11.5.6 Alternative Approaches to Aliasing

Euclid In is interesting to compare the approach to parameter passing taken here with that of 'conventional' language Euclid [LGH+78], which is the basis of the approach used in the Ada subset SPARK [CJM+92, CGM92].

In Euclid, the formal parameters have mode `var` or `nonvar`. Parameters of mode `var` may be assigned within the procedure; the actual parameter must be a variable. Parameters of mode `nonvar` are read-only within the procedure; any expression may be used as the actual parameter. The global variables are declared in a similar way, each procedure declaration including a list of the global variables which may be used within the procedure, with a `var` or `nonvar` mode for each such variable.

The following two requirements ensure the soundness of the proof rule used.

1. Reference passing is used to implement `var` parameters while `nonvar` are implemented by value passing (i.e. copy in).

2. The `var` actual parameters and the `var` global variables must be distinct. This less restrictive than Back's side condition.

In comparison to the approach adopted here, there are two key features. First, the global variables visible in the procedure body are listed in its specification. Second, global variables are assigned *modes* restricting some to be read-only.

In SPARK, the rules are more complicated because the Ada language (of which SPARK is a strict subset) does not specify the mechanism used to implement the different parameter modes, so that when a variable is used as the actual parameter for a `nonvar` parameter (written as `in` in the Ada syntax) the compiler is free to use pass by reference.

Adapting the Euclid Approach The anti-aliasing condition on the use of reference parameters proposed for the language of refinement is necessary because our language does not have the two key features of the Euclid approach: lists of visible global variables and global variable modes. Obviously, the Euclid approach could be adopted directly, by adding the global lists to the syntax.

In a refinement language it is also possible to adopt the Euclid approach without additional syntax: the global lists and modes could be *inferred*:

(i) the global variables are the free variables of the specification of a procedures, and

(ii) the `var` variables are the free variables which are assigned in the specification of a procedure (i.e. the frame variables).

To make this equivalent to the Euclid approach, the free variables and the assigned variables must be checked for a refinement, changing the refinement relationship so that if $S \sqsubseteq T$ then

(i) the free variables of T must be a subset of those of S, and

(ii) the assigned variables of T must be a subset of those of S.

This change would disallow a refinement such as **skip** $\sqsubseteq x := x$. Euclid introduces the annotations precisely because it is not convenient to analyse the implementation of a procedure in order to determine whether a particular call to the procedure causes aliasing. In a refinement language, the procedure specification can be analysed instead, provided that the properties of interest are maintained by refinement.

11.6 Semantics of Statements

The meaning of each form of statements is specified by its weakest precondition predicate transformer, as described in Chapter 2. Section 2.2.4 describes how the strongest postcondition predicate transformers are defined.

$$
\begin{array}{lcl}
wp_\rho\,\textbf{skip}\ Q & = & Q \\
wp_\rho\,\textbf{abort}\ Q & = & \mathbf{f} \\
wp_\rho(l_1,\ldots,\ l_n := t_1,\ldots,t_n)\ Q & = & Q\ asgn\ \langle l_1,\ldots,l_n\rangle\ \langle t_1,\ldots,t_n\rangle \\
wp_\rho(l_1,\ldots,\ l_n := \textbf{any}\ n\ P)\ Q & = & (\forall t_1 \ldots (\forall t_n \cdot P \Rightarrow \\
 & & \quad Q^{+n} asgn\ \langle l_1^{+n},\ldots,l_n^{+n}\rangle \\
 & & \quad\quad \langle \underline{n-1},\ldots,\underline{0}\rangle \\
 & & \textbf{where}\ t_i = (type_inf_\rho\ l_i)^{+(i-1)} \\
wp_\rho(\textbf{pre}\ P\ \textbf{in}\ S)\ Q & = & P \wedge wp_\rho\ S\ Q \\
wp_\rho\ S;T\ Q & = & wp_\rho\ S\ (wp_\rho\ T\ Q) \\
wp_\rho(\textbf{if}\ \textbf{or}_i\ P_i \Longrightarrow S_i)\ Q & = & \bigvee_i P_i \wedge \bigwedge_i (P_i \Rightarrow wp_\rho\ S_i\ Q) \\
wp_\rho(\textbf{var}\ t|e\ \textbf{in}\ S)\ Q & = & (wp_{\rho+t}\ S\ Q^+)[0 := e^+] \\
wp_\rho(\textbf{con}\ E\ \textbf{in}\ S)\ Q & = & ((wp_{\rho+type_inf_\rho E}\ S\ Q^+)[0 := E^+])^-
\end{array}
$$

Figure 11.11: Weakest Preconditions

For most statements the weakest precondition is *defined*. However, the loop statement (see Section 11.8), the recursion statement (see Section 11.9) and the statements associated with procedures (see Section 11.7) are defined by equivalences and the predicate transformers are calculated.

As described in Section 2.2.1, compound statements must be monotonic with respect to the refinement of their component statements. For weakest preconditions which are defined, monotonicity must be proved. For weakest preconditions calculated from an equivalence, monotonicity follows from the monotonicity of the statements used in the equivalence.

11.6.1 Semantic Functions

Weakest Preconditions The weakest precondition function is declared as:

$$wp_\rho : \mathcal{S} \to \mathcal{T} \to \mathcal{T}$$

For $wp_\rho\ s\ q$ to be a well-formed predicate, the statement s and the postcondition q must be well-formed, and q must be a predicate such that (for some α):

$$stmt_ok_{\rho,\alpha}\ s \wedge type_ok_\rho\ q \wedge bool_term_\rho\ q$$

Strongest Postcondition The strongest postcondition function is declared as:

$$sp_\rho : \mathcal{S} \to \mathcal{T} \to \mathcal{T}$$

The weakest preconditions (see Figure 11.11) and the strongest postconditions (see Figure 11.12) of simple statement forms are standard and ensure that compound statements are monotonic. The standard forms have been re-expressed using depth-indices.

$$
\begin{array}{lcl}
sp_\rho\,\textbf{skip}\;Q & = & Q \\
sp_\rho\,\textbf{abort}\;Q & = & \textbf{f} \\
sp_\rho(l_1,\ldots,\;l_n := t_1,\ldots,t_n)\;Q & = & \\
\quad(\exists u_1\ldots(\exists u_n\cdot Q^{+n}\;asgn\;\langle l_1^{+n},\ldots,l_n^{+n}\rangle\;\langle\underline{n-1},\ldots,\underline{0}\rangle\wedge & & \\
\qquad\qquad\qquad \bigwedge_i(l_i^{+n}=t_i^{+n}\;asgn\;\langle l_1^{+n},\ldots,l_n^{+n}\rangle\;\langle\underline{n-1},\ldots,\underline{0}\rangle))\ldots) & & \\
\quad\textbf{where}\;u_i=(type_inf_\rho\,l_i)^{+(i-1)} & & \\
sp_\rho(l_1,\ldots,\;l_n := \textbf{any}\;n\;P)\;Q & = & \\
\quad(\exists t_1\ldots(\exists t_n\cdot & & \\
\quad Q^{+n}\;asgn\;\langle l_1^{+n},\ldots,l_n^{+n}\rangle\;\langle\underline{n-1},\ldots,\underline{0}\rangle\wedge & & \\
\quad P\;asgn\;(\langle l_1^{+n},\ldots,l_n^{+n}\rangle\frown\langle\underline{n-1},\ldots,\underline{0}\rangle) & & \\
\qquad\quad(\langle\underline{n-1},\ldots,\underline{0}\rangle\frown\langle l_1^{+n},\ldots,l_n^{+n}\rangle))\ldots) & & \\
\quad\textbf{where}\;t_i=(type_inf_\rho\,l_i)^{+(i-1)} & & \\
sp_\rho(\textbf{pre}\;P\;\textbf{in}\;S)\;Q & = & sp_\rho\,S\;(P\wedge Q) \\
sp_\rho\,S;T\;Q & = & sp_\rho\,T\;(sp_\rho\,S\;Q) \\
sp_\rho(\textbf{if}\;\textbf{or}_i\;P_i\Longrightarrow S_i)\;Q & = & \bigvee_i sp_\rho\,S_i\;(P_i\wedge Q) \\
sp_\rho(\textbf{var}\;t|e\;\textbf{in}\;S)\;Q & = & (\exists t\cdot sp_{\rho+t}\,S\;(Q^+\wedge\underline{0}=e^+)) \\
sp_\rho(\textbf{con}\;E\;\textbf{in}\;S)\;Q & = & (\exists t\cdot sp_{\rho+t}\,S\;(Q^+\wedge\underline{0}=E^+)) \\
 & & \textbf{where}\;t=type_inf_\rho\,E
\end{array}
$$

Figure 11.12: Strongest Postconditions

Abstract Assignment The weakest preconditon of the abstract assignment statement makes this form of assignment equivalent to the demonic miraculous assignment of [BW89b], where the following definition of the weakest precondition is given (in identifier syntax)[10]:

$$
\begin{array}{l}
wp\;(x_1,\ldots,x_n := \textbf{any}\;y_1,\ldots,y_n\;\textbf{where}\;P)\;Q = \\
\qquad(\forall f_1,\ldots,f_n\cdot P[y_1,\ldots,y_n:=f_1,\ldots,f_n]\Rightarrow Q[x_1,\ldots,x_n:=f_1,\ldots,f_n])
\end{array}
$$

This definition requires the bound variables y_i to be renamed to fresh variables f_i to avoid the possibility that variables of the same name which are free in the preconditon Q are captured by the quantification.

If Statement A statement of the form **if** ... **fi** which can be used to build-up a standard conditional statement is proposed in [Mor88a]. However, this statement is not monotonic in its subcomponents and therefore cannot be used in our approach. (See also [BW89b]).

[10]This definition assumes that the left hand side of the assignment are simple variables and does not specify the type of the quantified variables.

Logical Constants The (**con** ...) statement is a deterministic form of the logical constant given in [MG90] using syntax with variables:

$$wp\ (\textbf{con}\ c\ \textbf{in}\ S)\ P \mathrel{\hat=} (\exists c \cdot wp\ S\ P))$$

This statement is *angelically* deterministic and therefore not conjunctive. Our version can be expressed using either the general form of logical constant, or the variable declaration.

$$\begin{array}{l}(\textbf{con}\ c \mathrel{\hat=} E \\ \ \textbf{in}\ S)\end{array} \equiv \begin{array}{l}(\textbf{con}\ c \\ \ \textbf{in}\ \{\ c = E\ \}; S)\end{array} \equiv \begin{array}{l}(\textbf{var}\ c \\ \ \textbf{in}\ c := E; S)\end{array}$$

The weakest precondition given above can be calculated, using the syntax with variables, from any of these equivalence. For example:

$$\begin{array}{ll} wp\ (\textbf{con}\ c \mathrel{\hat=} E\ \textbf{in}\ S)\ Q & \\ \quad = (\exists c \cdot wp\ (c = E; S)\ Q) & \text{[first equivalence]} \\ \quad = (\exists c \cdot c = E \wedge wp\ S\ Q) & \text{[wp of assert \& seq. comp.]} \\ \quad = (wp\ S\ Q)[c := E] & \text{[one point rule]} \end{array}$$

The general form of the logical constant does not have a strongest postcondition since it is not conjunctive. We can justify the definition given above using the second equivalence:

$$\begin{array}{ll} sp\ (\textbf{con}\ c \mathrel{\hat=} E\ \textbf{in}\ S)\ Q & \\ \quad = sp\ (\textbf{var}\ c\ \textbf{in}\ c := E; S)\ Q & \text{[second equivalence]} \\ \quad = (\exists c \cdot sp\ (c := E; S)\ Q) & \text{[sp of var decl]} \\ \quad = (\exists c \cdot sp\ S\ (sp\ c := E\ Q)) & \text{[sp of seq. comp.]} \\ \quad = (\exists c \cdot sp\ S\ (c = E \wedge Q)) & \text{[sp of assert]} \end{array}$$

11.6.2 Substitution for Assignment

The specifications of the weakest precondition and strongest postcondition for the two forms of assignment use substitution: in the assignment $x := e$, the value e substitutes for the variable x. Because of this connection between assignment and substitution, the syntax for substitution resembles an assignment: the substitution of the value e for the variable x is written $[x := e]$.

To transform an assignment to a substitution, the following complications must be taken into account:

(i) the left hand side of an assignment is a left value, rather than a variable,

(ii) the same variable may occur more than once on the left hand side of an assignment.

Statement	Equivalence
$a\ 1 := x$	$a := a \oplus (1, x)$
$a\ 1, a'\ 2 := x, y$	$a, a' := a \oplus (1, x), a' \oplus (2, y)$
$aa\ t_1\ t_2, aa\ u_1\ u_2 := x, y$	$a :=$ **if** $t_1 = u_1$ **then** $aa \oplus (t_1, aa\ t_1 \oplus (t_2, x) \oplus (u_2, y))$ **else** $aa \oplus (t_1, aa\ t_1 \oplus (t_2, x)) \oplus (u_1, aa\ u_1 \oplus (u_2, y))$
$one\ d := x$	$d := c_one\ x\ (two\ d)\ (three\ d)$
$one\ d, two\ d := x, y$	$d := c_one\ x\ y\ (three\ d)$
$one\ d\ t,$ $one\ d\ u := x, y$	$d := c_one\ (one\ d \oplus (t, x) \oplus (u, y))$ $(two\ d)\ (three\ d)$
$one\ (ad\ t),$ $one\ (ad\ u) := x, y$	$d := ad \oplus (t, c_one\ x\ (two\ (ad\ t))\ (three\ (ad\ t)))$ $\oplus (u, c_one\ y\ (two\ (ad\ t))\ (three\ (ad\ t)))$
$one\ (ad\ t),$ $two\ (ad\ u) := x, y$	$ad :=$ **if** $t = u$ **then** $ad \oplus (t, c_one\ x\ y\ (three\ (ad\ t)))$ **else** $ad \oplus (t, c_one\ x\ (two\ (ad\ t))\ (three\ (ad\ t)))$ $\oplus (u, c_one\ (one\ (ad\ t))\ y\ (three\ (ad\ t)))$

Figure 11.13: Transformation of Assignment Statements

In the rest of this section, the transformation to a substitution is described by a transformation to an equivalent assignment, which satisfies the rules for a substitution with only simple variables, not repeated, on the left hand side. Figure 11.13 shows some example assignment statements and their equivalences, using the variable and type declarations of page 155.

Assignment to Functions and Maps Both these forms are transformed in a similar way using the function or map override operator. An assignment with the form $f\ a := t$ where f is a variable of function type is equivalent to $f := f \oplus (a, t)$. The override operator for functions is `<+>` (see section B.1) while for maps it is `[+]` (see section B.3).

If the same variable occurs more than once in an assignment such as $f\ a, f\ b := t, u$, the equivalent assignment is either:

$$f := f \oplus (a, t) \oplus (b, u) \quad \text{or} \quad f := f \oplus (b, u) \oplus (a, t)$$

The two transformations are equivalent since $a \neq b$, which is a condition for the original assignment to be well-formed. A further complexity arises when two or more left values with two or more arguments are used in an assignment:

$$f\ a_1\ a_2, f\ b_1\ b_2 := t, u$$

This assignment is well-formed provided that $a_1 \neq b_1 \vee a_2 \neq b_2$. If $a_1 = b_1$ then the

equivalent assignment is:

$$f := f \oplus (a_1, f\ a_1 \oplus (a_2, t) \oplus (b_2, u))$$

while if $a_1 \neq b_1$ then it is:

$$f := f \oplus (a_1, f\ a_1 \oplus (a_2, t)) \oplus (b_1, f\ b_1 \oplus (b_2, u))$$

Since the correct choice cannot be determined statically, a conditional term is used, as shown in figure 11.13[11].

Assignment to Datatypes These are transformed using the datatype constructor for the appropriate case to form an updated datatype value. Thus $one\ d := x$ is equivalent to $d := c_one\ x\ (two\ d)\ (three\ d)$. Note that the datatype accessors which are not used as selectors on the left hand side of the assignment are used in the equivalent assignment. Since aliasing of datatype accessors is decidable, conditional terms are not required.

Assignment to Compound Types Some examples of complex left values involving functions and datatypes are shown in figure 11.13. The final example is noteworthy since it requires a conditional term in the equivalent assignment even though there is no disjointness condition.

The *asgn* Function In the specification of *wp* and *sp*, the function *asgn* is used to create the appropriate substitution. We write:

$$P\ asgn\ \langle l_1, \ldots, l_n \rangle\ \langle t_1, \ldots, t_n \rangle$$

for the predicate P modified by the substitution corresponding to the assignment of terms t_i to the left values l_i. A substitution is represented as a set of pairs (i, t) where i is an index and t is a term.

$$asgn : \mathbf{seq}\,\mathcal{L} \to \mathbf{seq}\,\mathcal{T} \to \mathbb{P}(\mathbb{N} \times \mathcal{T})$$

[11] A disadvantage of this approach is that the complexity of the equivalent rises rapidly. For the example shown there are two alternatives in the equivalence, for $f\ a_1\ a_2, f\ b_1\ b_2, f\ c_1\ c_2 := t, u, v$ there are five alternatives; adding a fourth assignment yields fifteen alternatives. Alternative approaches include (i) more restrictive well-formation rules (ii) allowing updates to functions of one argument only, i.e. using (in Pascal syntax) `ARRAY [1..m, 1..n] OF T` rather than `ARRAY [1..m] OF ARRAY [1..n] OF T`.

11.7 Semantics of Procedure Statements

11.7.1 Procedure Declaration and Call

The following definition of the meaning of a procedure declaration is standard (adapted to the variable-free syntax).

$$(\mathbf{proc}\ S\ \mathbf{in}\ T) \mathrel{\hat{=}} (T[0 := S^+])^-$$

The predicate transformers for (**proc** ...) are calculated by first making the substitution, eliminating all free statement variables. The weakest precondition and strongest postcondition are required for both procedure blocks and free statement variables – see Section 12.1.2 – in the second case the substitution is achieved using the environment.

$$\begin{array}{lcl} wp_\rho(\mathbf{proc}\ S\ \mathbf{in}\ T)\ Q & = & wp_\rho(T[0 := S^+])^- \\ wp_\rho\ \underline{I} & = & wp_\rho\ S\ \mathbf{where}\ \rho\ !!\ I = Proc\ S \end{array}$$

$$\begin{array}{lcl} sp_\rho(\mathbf{proc}\ S\ \mathbf{in}\ T)\ Q & = & sp_\rho(T[0 := S^+])^- \\ sp_\rho\ \underline{I} & = & sp_\rho\ S\ \mathbf{where}\ \rho\ !!\ I = Proc\ S \end{array}$$

Theorem 11.1 demonstrates the monotonicity of this semantics of procedures.

Theorem 11.1 Monotonicity of statement substitution *For any statements T, T' not referring to the statement variable X and any statement S_X possibly referring to X:*

$$T \sqsubseteq T' \Rightarrow S[X := T] \sqsubseteq S[X := T]$$

A proof of this theorem is given in [Bac87, Theorem 1, p11].

It is possible to extend the definition of refinement to allow for statements with free statement variables (in this case, the refinement must hold for any substitution of the variables). However, this extension is not necessary in the approach adopted here.

11.7.2 Parameters

The well-formation rules ensure that abstraction and application match. Therefore the meaning of an application can be defined for each form of abstraction. Using the identifier syntax, the definitions are:

$$\begin{array}{lcl} [\ \mathbf{val}\, x : t\ ]\ S\ [\ e\ ] & \hat{=} & (\mathbf{var}\ x : t\ \mathbf{in}\ x := e; S) \\ [\ \mathbf{res}\, x : t\ ]\ S\ [\ y\ ] & \hat{=} & (\mathbf{var}\ x : t\ \mathbf{in}\ S; y := x) \\ [\ \mathbf{valres}\, x : t\ ]\ S\ [\ y\ ] & \hat{=} & (\mathbf{var}\ x : t\ \mathbf{in}\ x := y; S; y := x) \\ [\ \mathbf{ref}\ x : t\ ]\ S\ [\ y\ ] & \hat{=} & S[x := y] \end{array}$$

It is shown in [Bac87] that the **val** and **res** parameter passing mechanisms defined in this way are monotonic, subject to renaming variables to prevent capture. The conditions for monotonicity of pass-by-reference have been discussed in Section 11.5. Variable renaming is avoided by using the variable-free syntax:

$$\begin{array}{lcl}
[\ \mathbf{val}\, t\ ]\ S\ [\ e\ ] & \hat{=} & (\mathbf{var}\ t\ \mathbf{in}\ \underline{0} := e^{+}; S) \\
[\ \mathbf{res}\, t\ t_i\ ]\ S\ [\ y\ ] & \hat{=} & (\mathbf{var}\ t\ \mathbf{in}\ \underline{0} := t_i^{+}; S; y^{+} := \underline{0}) \\
[\ \mathbf{valres}\, t\ ]\ S\ [\ y\ ] & \hat{=} & (\mathbf{var}\ t\ \mathbf{in}\ \underline{0} := y^{+}; S; y^{+} := \underline{0}) \\
[\ \mathbf{ref}\ t\ ]\ S\ [\ y\ ] & \hat{=} & (S[0 := y^{+}])^{-}
\end{array}$$

The weakest preconditions can be calculated from these definition giving results which are the same as the definitions given in [Mor88c]. However, it should be noted that y in the definitions above is not restricted to a variable but can be a left-value.

11.7.3 Multiple Parameters

An abstraction with multiple parameters is defined by the *simultaneous* substitution of the list of parameters. (An abstraction with multiple parameters cannot be described using nested abstraction with single parameters because this results in sequential substitution.)

Let t, u, v, w and e be terms, with left values x, y and z which do not alias. The application of $[e, x, y, z]$ to an abstraction with multiple parameters is defined as follows:

$$\begin{array}{lcl}
[\ \mathbf{val}\, t, \mathbf{valres}\, u, \mathbf{res}\, v\ v_i, \mathbf{ref}\ w\ ] & & (\mathbf{var}\ t; u; v; w\ \mathbf{in} \\
\quad S\ [\ e, x, y, z\ ] & \hat{=} & \quad \underline{3}, \underline{2}, \underline{1} := e^{+4}, x^{+4}, v_i^{+2}; \\
 & & \quad S[0 := z^{+4}]; \\
 & & \quad x^{+4}, y^{+4} := \underline{2}, \underline{1})
\end{array}$$

The equivalent statement has the following components:

(i) a copy-in assignment for the **val** and **valres** parameters,

(ii) initialisation of the **res** parameter,

(iii) a substitution for the **ref** parameters, and

(iv) a copy-out assignment for the **res** and **valres** parameters.

11.7.4 Refinement of Parameterised Statements

Following [Bac87] we extend refinement of statements to parameterised statements.

Definition 11.1 *Let S, S' be statements and* **abs** *stand for one of the forms of parameterisation.*

$$[\ \mathbf{abs}\ t\ ]\ S \sqsubseteq [\ \mathbf{abs}\ u\ ]\ S'$$

if and only if

$$[\ \mathbf{abs}\ t\ ]\ S\ [\ e\ ] \sqsubseteq [\ \mathbf{abs}\ u\ ]\ S'\ [\ e\ ]$$

for all expressions e (for which the application is well-formed).

We make the additional restriction that for any refinement of parameterised statements the two lists of formal parameters must be identical, simplifying the above equivalence to $(\forall t \cdot S \sqsubseteq S')$. Refinement of a parameterised statement occurs implicitly in recursion (see Section 11.9), when the parameters are necessarily identical. Since the syntax prevents an abstraction from being labelled[12] (see Section A.4.2) the explicit refinement of a parameterised statement is restricted.

11.8 Semantics of the Loop Statement

The loop statements is defined to be equivalent to an abstract assignment under a precondition.

$$
\begin{array}{l}
(\mathbf{while}\ I\ \mathbf{vary}\ E\ \mathbf{loop}\ \mathbf{or}_i\ B_i \Longrightarrow S_i)\ \equiv \\
\quad (\mathbf{pre} \\
\qquad I \wedge \\
\qquad \mathbf{All}_\rho\ \langle x_1, \ldots, x_n \rangle\ \bigwedge_i (I \wedge B_i \Rightarrow ((wp_{\rho+\mathbb{Z}}\ S_i^+\ (I^+ \wedge E^+ < \underline{0}))[\underline{0} := E^+])^-) \wedge \\
\qquad \mathbf{All}_\rho\ \langle x_1, \ldots, x_n \rangle\ (I \wedge \bigvee_i B_i \Rightarrow 0 < E) \\
\quad \mathbf{in} \\
\qquad \underline{x_1}, \ldots, \underline{x_n} := \mathbf{any}\ n\ (I \wedge \bigwedge_i \neg B_i)^{+n}[x_1 + n, \ldots, x_n + n := \underline{n-1}, \ldots, \underline{0}])
\end{array}
$$

where $\langle x_1, \ldots, x_n \rangle$ is a sequence containing the indices of the variables assigned in any of the statements within the loop: $\bigcup_i \{\ \Psi\ S_i\ \}$. The principles for the equivalent statements are:

1. The precondition ensures that the invariant and the variant of the loop have the following properties (standard from Hoare logic):

   (a) the invariant holds initially,

   (b) in all states in which the invariant and one of the guards B_i holds, the corresponding statement S_i maintains the invariant and decreases the variant, and

[12] The intention of the syntax was to prevent explicit refinement of parameterised statements (which appears to have no purpose in our approach), but it is not effective. This suggests that the restriction should be tightened or dropped.

(c) in all states in which the invariant and any of the guards hold, the variant is greater than zero.

2. The abstract assignment updates all the variables which may be assigned by the statements in the loop in any way which maintains the invariant of the loop and makes all the loop guards false.

Monotonicity The monotonicity of the loop statement follows from the monotonicity of the statements used in the equivalence.

Continuity When a loop contains a non-continuous statement an integer variant may not exist (even though the loop terminates). Consider, for example, the following program adapted from Boom's paper [Boo82], where `i:Nat` and `first:Bool`:

```
first := True ;
WHILE i >= 0 VARY ?
LOOP
  first ==> i := WHERE True ; first := False
OR
  not first && i > 0 ==> i := i - 1
END LOOP
```

Since there is no variant function which is always decreased by the first iteration of the loop, the precondition of the equivalent statement is false and the loop behaves as **abort**.

The only primitive statement in the language which can be non-continuous is the non-deterministic assignment when the set of values satisfying the predicate is infinite. (Back and Von Wright [BW98, §22] characterise statements which may be non-continuous.) The requirement for an integer loop variant prevents some uses of infinite non-deterministic assignments in loop bodies. This restriction could be relaxed by allowing a variant of any well-founded set, including ordinals, to be used to show termination of a loop. We note that the standard rule for introducing a loop in the refinement calculus (see below) also assumes a variant from $\mathbb{N}$.

11.8.1 Frame Variables

The variables which are assigned (and therefore *may* be changed) by a statement are called the *frame* variables. We write $\Psi\, S$ to stand for the set of such variables.

$$\begin{array}{lcl}
\Psi(l_1,\ldots,l_n := t_1,\ldots,t_2) & = & \bigcup_i \mathit{lhs}\, l_i \\
\Psi(l_1,\ldots,l_n := \mathbf{any}\ n\ t) & = & \bigcup_i \mathit{lhs}\, l_i \\
\Psi(\mathbf{var}\ t_1|t_2\ \mathbf{in}\ S) & = & \{\ i-1|i \in \Psi\, S \wedge i > 0\ \} \\
\Psi(\mathbf{con}\ t\ \mathbf{in}\ S) & = & \{\ i-1|i \in \Psi\, S \wedge i > 0\ \} \\
\Psi(\mathbf{proc}\ S_1\ \mathbf{in}\ S_2) & = & \{\ i-1|i \in \Psi\, S_2[0 := S_1^+]\ \} \\
\Psi(\mathbf{rec}\ S_1\ \mathbf{vary}\ t\ \mathbf{in}\ S_2) & = & \{\ i-1|i \in \Psi\, S_2[0 := S_1^+]\ \}
\end{array}$$

Other cases are defined in the obvious way. Since the definition of Ψ for **proc** and **rec** substitutes the statement variable, no case is required for the free statement variable $\underline{I}$ and abstraction and applications are paired:

$$\begin{array}{lcl}
\Psi([\ \mathbf{val}\, t\ ]\ S\ [\ e\ ]) & = & \{\ i-1|i \in \Psi\, S \wedge i > 0\ \} \\
\Psi([\ \mathbf{valres}\, t\ ]\ S\ [\ l\ ]) & = & \{\ \mathit{lhs}\, l\ \} \cup \{\ i-1|i \in \Psi\, S \wedge i > 0\ \} \\
\Psi([\ \mathbf{res}\, t\ t_i\ ]\ S\ [\ l\ ]) & = & \{\ \mathit{lhs}\, l\ \} \cup \{\ i-1|i \in \Psi\, S \wedge i > 0\ \}
\end{array}$$

The definition of Ψ for the **ref** parameter mode is the same as that for **valres**.

11.8.2 All State Quantification

The properties of the statements in the loop must hold *in all states*. This could be represented by universal quantification over *all* the variables, however, it is sufficient to quantify over the variables which are assigned in the loop body. The quantified term is constructed by the function $\mathbf{All}_\rho : \mathbf{seq}\,\mathbb{N} \to \mathcal{T} \to \mathcal{T}$, with the necessary adjustments to the indices.

$$\begin{array}{lcl}
\mathbf{All}_\rho\ \langle x_1,\ldots,x_n\rangle\ p & = & (\forall t_1 \ldots \forall t_n \cdot p^{+n}[x_1+n,\ldots,x_n+n := \underline{n-1},\ldots,\underline{0}]) \\
 & & \mathbf{where}\ t_i = (\mathit{type_inf}_\rho\ \underline{x_i})^{+(i-1)}
\end{array}$$

11.8.3 Loop Predicate Transformers

The predicate transformers for the loop statement can be calculated from the equivalence. Let:

$$\begin{array}{lcl}
\mathbf{loop} & \hat{=} & \mathbf{while}\ I\ \mathbf{vary}\ E\ \mathbf{loop}\ \mathbf{or}_i\ B_i \Longrightarrow S_i \\
xs & \hat{=} & \text{a sequence of the indices of the assigned variables } \bigcup_i \{\ \Psi\, S_i\ \} \\
A_1 & \hat{=} & \mathbf{All}_\rho\ xs\ \bigwedge_i (I \wedge B_i \Rightarrow ((wp_{\rho+\mathbb{Z}}\, S_i^+\ (I^+ \wedge E^+ < \underline{0}))[\underline{0} := E^+])^-) \\
A_2 & \hat{=} & \mathbf{All}_\rho\ xs\ (I \wedge \bigvee_i B_i \Rightarrow 0 < E)
\end{array}$$

then:

$$
\begin{array}{ll}
wp_\rho \ \textbf{loop}\ Q & \\
= I \wedge A_1 \wedge A_2 \wedge wp_\rho\ (\underline{x_1}, \ldots, \underline{x_n} := & \text{[wp of the } \textbf{pre} \text{]} \\
\quad \textbf{any}\ n\ (I \wedge \bigwedge_i \neg B_i)^{+n} & \\
\quad\quad [x_1 + n, \ldots, x_n + n := \underline{n-1}, \ldots, \underline{0}]))\ Q & \\
= I \wedge A_1 \wedge A_2 \wedge \mathbf{All}_\rho\ xs\ (I \wedge \bigwedge_i \neg B_i \Rightarrow Q) & \text{[wp of the abstract assign.]}
\end{array}
$$

$$
\begin{array}{ll}
sp_\rho \ \textbf{loop}\ P & \\
= sp_\rho\ \ (\underline{x_1}, \ldots, \underline{x_n} := \textbf{any}\ n\ (I \wedge \bigwedge_i \neg B_i)^{+n} & \text{[sp of the } \textbf{pre} \text{]} \\
\quad [x_1 + n, \ldots, x_n + n := \underline{n-1}, \ldots, \underline{0}])\ (P \wedge I \wedge A_1 \wedge A_2) & \\
= (\exists t_1 \ldots \exists t_n \cdot & \text{[sp of abstract assignment]} \\
\quad (P \wedge I \wedge A_1 \wedge A_2)^{+n}[x_1 + n, \ldots, x_n + n := \underline{n-1}, \ldots, \underline{0}] \wedge & \\
\quad (I \wedge \bigwedge_i \neg B_i)^{+n}[x_1 + n, \ldots, x_n + n := \underline{n-1}, \ldots, \underline{0}] & \\
\quad\quad [x_1 + n, \ldots, x_n + n, n-1, \ldots, 0 := \underline{n-1}, \ldots, \underline{0}, \underline{x_1 + n}, \ldots, \underline{x_n + n}]) & \\
\quad \textbf{where}\ t_i = (type_inf_\rho\ \underline{x_i})^{+(i-1)} & \\
= A_1 \wedge A_2 \wedge I \wedge \bigwedge_i \neg B_i \wedge & \text{[quantified variables not free]} \\
\quad (\exists t_1 \ldots \exists t_n \cdot & \\
\quad\quad (P \wedge I)^{+n}[x_1 + n, \ldots, x_n + n := \underline{n-1}, \ldots, \underline{0}]) &
\end{array}
$$

This completes the definition of the loop statement; to justify this definition we show below that a loop statement defined in this way can be implemented by a standard iteration statement.

11.8.4 Comparison with the Refinement Calculus

The similarity between the approach described here and that used in Abrial's Abstract Machine Notation has already been noted. The connection with the treatment of loops in Morgan's refinement calculus, apparently very different, is described here. The refinement calculus law for introducting an iteration [MR87, Law 11], is:

$$
\frac{\vec{w} : [true,\ inv,\ \neg(\bigvee i : I \cdot G_i)]}{\begin{array}{l}\textbf{do} \\ \quad ([]\ i : I \cdot G_i \rightarrow \vec{w} : [G_i,\ inv,\ 0 \leq var < var_0]) \\ \textbf{od}\end{array}}
$$

The loop with explicit invariant and variant can be seen as an alternative syntax for this refinement law, which has no side-conditions. The properties of the law are:

1. The loop refines a statement which establishes the invariant and the disjunction of the guards.

2. When the corresponding guard applies, each statement within the loop must be a refinement of the statement which maintains the invariant and decreases the variant.

These are properties of the loop equivalence given above[13].

11.8.5 Implementing the Loop Statement

The following theorem shows that the loop statement can be implemented using the standard **do** ... **od** statement, by discarding the invariant and variant.

Theorem 11.2 Implementing Loops *For any invariant predicate I, variant expression E, guards B_i and statements S_i, let:*

$$\begin{aligned} \textbf{loop} &\mathrel{\widehat{=}} (\textbf{while } I \textbf{ vary } E \textbf{ loop or}_i\ B_i \Longrightarrow S_i) \\ \textbf{dood} &\mathrel{\widehat{=}} \textbf{do} \mathop{[\!]}_{i} B_i \Longrightarrow S_i \textbf{od} \end{aligned}$$

then **loop** $\sqsubseteq$ **dood**. *The proof is on page 220.*

11.9 Semantics of Recursion

We define the recursion block by an equivalence:

$$\begin{aligned} &(\textbf{rec } [\ a_1, \ldots, a_n\ ]\ S &\mathrel{\widehat{=}}\ & [\ a_1, \ldots, a_n\ ]\ (\textbf{con } t_v \textbf{ in} \\ &\quad \textbf{vary } t_v \textbf{ in } T) & & \quad (\textbf{proc } [\ b_1, \ldots, b_n\ ]\ \{\ 0 \le t'_v \land t'_v < \underline{n}\ \}; S' \\ & & & \qquad \textbf{in } T)) \\ & & & \textbf{where } b_i = a_i^{+(n+1).(i-1)} \\ & & & \qquad\quad t'_v = t_v^{+(n+1).n} \\ & & & \qquad\quad S' = S^{+(n+1).n} \end{aligned}$$

provided that

$$S \sqsubseteq (\textbf{con } t_v \textbf{ in } (T[0 := \{0 \le t'^{+}_v \land t'^{+}_v < \underline{n+1}\}; S'^{+}])^{-})$$

In the following text, this equation is referred to as the *proviso*. The proviso expresses the constraint on the statement ($\textbf{rec } S \textbf{ vary } t_v \textbf{ in } T$) that the statement T must be

[13]The only different is that the equivalence is based on the traditional form of the invariance proof rule for loops, while Morgan uses an alternative. The alternative requires that the invariant and the guard of each statement imply the precondition for the statement to establish a variant value greater than or equal to zero. This is almost the same as the third conjunct in the precondition of the loop equivalence, which requires that the disjunction of the guards implies that the variant is greater than zero. If this is not the case in the refinement calculus version, then at least one statement in the loop body is infeasible.

a refinement of S. Since the syntax ensures that S is the specification of T, it is not necessary to check the proviso explicitly: it is checked with the rest of the refinement text. We make this argument more formal in Section 12.2.

In the equivalent statement for the recursion, a logical constant is introduced to hold the initial value of the variant and a procedure is introduced to specify the recursive statement variable. The specification of the procedure is the statement S which specifies the recursion, including the formal parameters. The procedure specification also has a precondition to show that the variant t_v is decreased, bounded by zero.

The scope of the different components of the equivalent statement needs to considered carefully. The logical constant is visible in the statement T as $\underline{1}$ and the recursive variable is $\underline{0}$. Within the procedure specification, the scope of S and t_v needs to be adjusted so that they depend on the parameters of the procedure. The scope adjustment makes use of an extension of the operation introduced in Section 9.2 for adjusting indices: for a term (or statement) t, $t^{+i,j}$ denotes the term t with indices greater than or equal to j increased by i. The default value for j is 0.

Recursion Specification The procedure specification part of the recursion equivalence is used in the specification of well-formation.

$$
\begin{array}{rcl}
\mathit{rec_spec}\ \langle a_1, \ldots, a_n \rangle\ S\ t_v & \widehat{=} & [\ b_1, \ldots, b_n\ ]\ (\{\ 0 \leq t'_v \wedge t'_v < \underline{n}\ \}; S') \\
\textbf{where}\ b_i & = & a_i^{+(n+1),(i-1)} \\
t'_v & = & t_v^{+(n+1),n} \\
S' & = & S^{+(n+1),n}
\end{array}
$$

Continuity As with a loop statement, when a recursion contains a non-continuous statement it may not be possible to find an integer variant. The use of recursion is restricted to cases for which an integer variant exists. Without a variant, the 'proviso' does not hold and the refinement fails. Equivalently, an invalid variant leads to a precondition of 'false' for (one of) the recursive call(s) and the recursion block aborts[14].

11.9.1 Recursion Predicate Transformers

We can calculate the weakest precondition from this definition, assuming that the proviso holds:

$$
\begin{array}{ll}
wp_\rho(\textbf{rec}\ S\ \textbf{vary}\ t_V\ \textbf{in}\ T)\ Q & \\
\quad = wp_\rho(\textbf{con}\ t_V\ \textbf{in}\ (\textbf{proc}\ \ldots))\ Q & [\text{expanding defn.}] \\
\quad = ((wp_{\rho'}(\textbf{proc}\ \{0 \leq t_V^+ \wedge t_V^+ < \underline{0}\}; S^+\ \textbf{in}\ T)\ Q^+)[0 := t_V^+])^- & [\text{wp for log con}] \\
\quad = ((wp_{\rho'}(T[0 := \{0 \leq t_V^{+2} \wedge t_V^{+2} < \underline{1}\}; S^{+2}])^-\ Q^+)[0 := t_V^+])^- & [\text{wp for proc}]
\end{array}
$$

[14] Unless the recursion is called in an infeasible context.

where $\rho' = \rho + type_inf_\rho\, t_v$

Similarly, assuming that the proviso holds:

$$
\begin{aligned}
&sp_\rho(\textbf{rec } S \textbf{ vary } t_V \textbf{ in } T)\ P = \\
&\quad (\exists u \cdot sp_\rho(T[\underline{0} := \{0 \le t_V^{+2} \wedge t_V^{+2} < \underline{1}\}; S^{+2}])^-\ (P^+ \wedge \underline{0} = t_V^+)) \\
&\quad \textbf{where } u = type_inf_\rho\, t_v,\ \rho' = \rho + u
\end{aligned}
$$

11.9.2 Introducing a Recursion

We show that a recursion block can be introduced wherever its specifying statement could be introduced.

Theorem 11.3 Introducing a Recursive Call *Let S_1, S_2 be statements in which the recursive variable $\underline{0}$ is not free, let T be a statement possibly containing $\underline{0}$ and let t_V be a variant. Then*

$$S_1 \sqsubseteq S_2 \Rightarrow S_1 \sqsubseteq (\textbf{rec } S_2 \textbf{ vary } t_V \textbf{ in } T)$$

provided that

$$S_2 \sqsubseteq (\textbf{con } t_V \textbf{ in } (T[\underline{0} := \{0 \le t_V^{+2} \wedge t_V^{+2} < \underline{1}\}; S^{+2}])^-))$$

The proof is on page 221. A trivial way to satisfy the proviso of this theorem is to make the statement in the scope of the recursive variable the same as the specifying statement. Theorem 11.3 then shows that:

$$S_1 \sqsubseteq S_2 \Rightarrow S_1 \sqsubseteq (\textbf{rec } S_2 \textbf{ vary } t_V \textbf{ in } S_2^{+2})$$

always holds.

11.9.3 Implementing Recursion

We show that a recursion block can be implemented using a simple recursion, by discarding the specifying statement and the variant. We write (μS) for recursion over the statement variable $\underline{0}$ in S.

Theorem 11.4 Implementing Recursion *For any S, t_V and T in a well-formed recursion statement:*

$$(\textbf{rec } S \textbf{ vary } t_V \textbf{ in } T) \sqsubseteq (\mu T)$$

provided that

$$S \sqsubseteq (\textbf{con } t_V \textbf{ in } (T[\underline{0} := \{0 \le t_V^{+2} \wedge t_V^{+2} < \underline{1}\}; S^{+2}])^-)$$

The proof is on page 221.

11.10 Context Propagation

In this section, the context propagation rules are specified, following the approach outlined in Section 2.2.6. These rules have already been used in the $stmt_cc_\rho$ specification in Figures 11.5, 11.6 and 11.8.

11.10.1 Context Rules

Figure 11.14 gives context rules for compound statements based on [Bac88, §8]. The context rule for primitive statements is given in Section 2.2.6.

Variable and Constant Block The initial context applies to the statement within the block. The variable or constant must be removed from the final context of the statement.

Loop In a loop, the context of the guarded command is strengthened by the guard and the invariant. The final context of the complete loop is its postcondition, which does not depend on the final context of each alternatives. Nor does the context following a loop depend directly on the initial context; however, the initial context may allow the loop invariant to be strengthened, thus strengthening the context of the alternatives.

Procedure The declaration of a procedure does not affect the context of the statement within the scope of the procedure. However, the context cannot be propagated to the specifying statement, since the procedure can be called in any other context. The final context Q can be determined by substituting the specifying statement S for the procedure variable.

Recursion Back's [Bac88, §8] context rule for refinement is conditional; it is not easy to use since it does not specify how a context predicate satisfying the condition can be found. The final context is determined in the same ways as the procedure statements, using the equivalent statement for the recursion. In the rule shown, the precondition $\{0 \leq E^+ \wedge E^+ < \underline{0}\}$ which is part of the equivalent statement is omitted: without it, the context is weaker but simpler.

Procedure Call and Parameters In the hypotheses of the rules for procedures and recursion given above, free statement variables are eliminated by substitution. The equivalences for application of parameters to abstraction statements given on page 183 are used to eliminate application and abstraction.

11.10.2 Following Context

The context annotation following a statement is given by the function $fcntx_\rho : \mathcal{T} \to \mathcal{S} \to \mathcal{T}$. For most forms of statement, the strongest postcondition is used. Two exceptions are a weaker (but simpler) context is chosen for the loop and recursion statements, as shown in the inference rule in Figure 11.14. Since statement variables are replaced by the statement which defines them, abstractions and applications are matched.

$$
\begin{array}{lcl}
fcntx_\rho\, p\, (\textbf{while}\ t_I\ \textbf{vary}\ t_V\ \textbf{loop}\ \textbf{or}_i\ t_i \Longrightarrow S_i) & & \\
 & = & t_I \wedge \bigwedge_i \neg t_i \\
fcntx_\rho\, p\, (\textbf{rec}\ S_1\ \textbf{vary}\ t\ \textbf{in}\ S_2) & = & fcntx_\rho\, p S_1 \\
fcntx_\rho\, p\, ([\ \textbf{val}\, t\ ]\ S\ [\ e\ ]) & = & (\exists t \cdot fcntx_{\rho+t}(p^+ \wedge \underline{0} = e^+)\ S)
\end{array}
$$

... similarly for other modes, based on the equivalences of Section 11.7.2

$$
fcntx_\rho\, p\, S \quad = \quad sp_\rho\, S\, p\ \textbf{otherwise}
$$

$$\text{(Seq)}\quad \frac{\{\ P\ \}S \equiv \{\ P\ \}S'\{\ Q\ \} \qquad \{\ Q\ \}T \equiv \{\ Q\ \}T'\{\ R\ \}}{\{\ P\ \}(S;T) \equiv \{\ P\ \}(S';T')\{\ R\ \}}$$

$$\text{(Pre)}\quad \frac{\{\ P \wedge t\ \}S \equiv \{\ P \wedge t\ \}S'\{\ Q\ \}}{\{\ P\ \}(\textbf{pre}\ t\ \textbf{in}\ S) \equiv \{\ P\ \}(\textbf{pre}\ t\ \textbf{in}\ \{\ P \wedge t\ \}S')\{\ Q\ \}}$$

$$\text{(If)}\quad \frac{\{\ P \wedge t_i\ \}S_i \equiv \{\ P \wedge t_i\ \}S_i'\{\ Q_i\ \}}{\{\ P\ \}(\textbf{if}\ \ \textbf{or}_i\ t_i \Longrightarrow S_i) \equiv \{\ P\ \}(\textbf{if}\ \ \textbf{or}_i\ t_i \Longrightarrow \{\ P \wedge t_i\ \}S_i')\{\ \bigvee_i Q_i\ \}}$$

$$\text{(Var)}\quad \frac{\{\ P^+ \wedge \underline{0} = e^+\ \}S \equiv \{\ P^+ \wedge \underline{0} = e^+\ \}S'\{\ Q\ \}}{\{\ P\ \}(\textbf{var}\ t|e\ \textbf{in}\ S) \equiv \{\ P\ \}(\textbf{var}\ t|e\ \textbf{in}\ \{\ P^+ \wedge \underline{0} = e^+\ \}S')\{\ (\exists t \cdot Q\ )\}}$$

$$\text{(Con)}\quad \frac{\{\ P^+ \wedge \underline{0} = e^+\ \}S \equiv \{\ P^+ \wedge \underline{0} = e^+\ \}S'\{\ Q\ \}}{\{\ P\ \}(\textbf{con}\ e\ \textbf{in}\ S) \equiv \{\ P\ \}(\textbf{con}\ e\ \textbf{in}\ \{\ P^+ \wedge \underline{0} = e^+\ \}S')\{\ (\exists\, type_inf_\rho\, e \cdot Q\ )\}}$$

$$\text{(Loop)}\quad \frac{\{\ t_I \wedge t_i\ \};S_i \equiv \{\ t_I \wedge t_i\ \}S_i'\{\ Q_i\ \}}{\{\ P\ \}(\textbf{while}\ t_I\ \textbf{vary}\ t_V\ \textbf{loop}\ \textbf{or}_i\ t_i \Longrightarrow S_i) \equiv \{\ P\ \}(\textbf{while}\ t_I\ \textbf{vary}\ t_V\ \textbf{loop}\ \textbf{or}_i\ t_i \Longrightarrow \{\ t_I \wedge t_i\ \}S_i')\ \{\ t_I \wedge \bigwedge_i \neg t_i\ \}}$$

$$\text{(Proc)}\quad \frac{\{\ P^+\ \}T[0 := S^+] \equiv \{\ P^+\ \}T'[0 := S^+]\{\ Q\ \}}{\{\ P\ \}(\textbf{proc}\ S\ \textbf{in}\ T) \equiv \{\ P\ \}(\textbf{proc}\ S\ \textbf{in}\ \{\ P^+\ \}T')\{\ Q^-\ \}}$$

$$\text{(Rec)}\quad \frac{T[0 := S^+] \equiv T'[0 := S^+]\{\ Q\ \}}{\{\ P\ \}(\textbf{rec}\ S\ \textbf{vary}\ t_V\ \textbf{in}\ T) \equiv \{\ P\ \}(\textbf{rec}\ S\ \textbf{vary}\ t_V\ \textbf{in}\ T')\{\ Q^-\ \}}$$

Figure 11.14: Specification of Context Propagation

Chapter 12

Refinement Texts

A refinement text records the stepwise development of a program from a specification. In this chapter, we show how a refinement text can be represented conveniently by adding a refinement *statement* to the abstract syntax of statements already described.

12.1 The Refinement Statement

A refinement statement has a pair of statements:

$$\mathcal{S} \ ::= \ \ldots \ | \ \mathcal{S} \ \textbf{ref} \ \mathcal{S}$$

Example Refinement Statement Let A_i be unspecified atomic statements in the statement E defined as:

$$E \widehat{=} A_0 \ \textbf{ref} \ (((A_1 \ \textbf{ref} \ A_2); A_3) \ \textbf{ref} \ ((A_4 \ \textbf{ref} \ (A_6; A_7)); A_5))$$

Statement E is illustrated in Figure 12.1 using the *refinement diagrams* of [Bac91]. Each refinement is shown as a double horizontal line with the statement being refined at the left end of the line and the refining statement at the right hand end. The left hand side of the refinement, which can be called the *specification* is often a simpler statement than its *implementation*, on the right hand side of the refinement. In stepwise development, further refinements are made to the implementation (i.e. on the right hand side) of a previous refinement. However, as shown in the example, it is also possible to have 'nested' refinement, where a compound statement created by a number of refinement steps becomes the specification of another refinement.

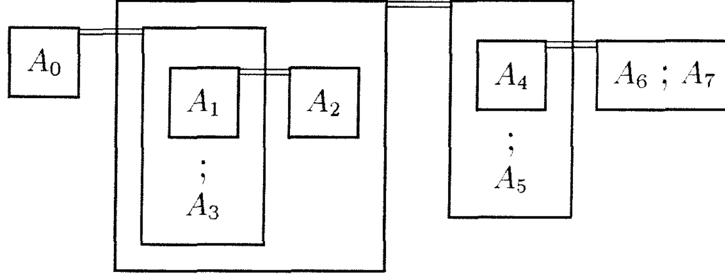


Figure 12.1: Refinement Diagram for Statement E

12.1.1 Specifications and Implementations

We define two functions on the syntax of statements: *spec* : $\mathcal{S} \rightarrow \mathcal{S}$ takes the left hand side of each refinement to give the least refined statement, while *imp* : $\mathcal{S} \rightarrow \mathcal{S}$ takes the right hand side to give the most refined statement. The definitions are in Figure 12.2. Applying these definitions to the example statement E obtains the specification and implementation:

$$spec\ E = A_0 \qquad imp\ E = (A_6; A_7); A_5$$

so that the overall refinement represented by E is $A_0 \sqsubseteq (A_6; A_7); A_5$.

12.1.2 Well-formation and Correctness

Since the refinement construct is represented as a statement, cases of the $stmt_ok_{\rho,\alpha}$ and $stmt_cc_\rho$ functions are required. The definitions are shown in Figure 12.3.

Well-formation A refinement is well-formed provided the component statements are well-formed. The formal parameters must be the same, as noted in Section 11.7.4.

Correctness The correctness of the refinement statement *includes* the condition for correct refinement. The term $imp\,S \sqsubseteq spec\,T$, used in the correctness condition, is expanded using the characterisation theorem, as described in Section 2.2.5. The structure of the $stmt_cc_\rho$ definition explains the cases required for the predicate transformers, used in the characterisation theorem:

(i) no cases of *wp* and *sp* are required for the refinement statement, since the arguments of $\sqsubseteq$ do not contain refinement statements,

(ii) cases of *wp* and *sp* are required for the call statement $\underline{I}$, since refinements can occur within the scope of a procedure declaration and a statement variable may be free on the left or right of $\sqsubseteq$.

$$
\begin{array}{lcl}
\mathit{spec}\,A & = & A \quad \textbf{where} \quad A \text{ is atomic} \\
\mathit{spec}(S;T) & = & \mathit{spec}\,S; \mathit{spec}\,T \\
\mathit{spec}(\textbf{pre}\ t\ \textbf{in}\ S) & = & (\textbf{pre}\ t\ \textbf{in}\ \ \mathit{spec}\,S) \\
\mathit{spec}(\textbf{if}\ \textbf{or}_i\ t_i \Longrightarrow S_i) & = & (\textbf{if}\ \textbf{or}_i\ t_i \Longrightarrow \mathit{spec}\,S_i) \\
\mathit{spec}\,(\textbf{while}\ t_I\ \textbf{vary}\ t_V & = & (\textbf{while}\ t_I\ \textbf{vary}\ t_V \\
\qquad\qquad \textbf{loop}\ \textbf{or}_i\ t_i \Longrightarrow S_i) & & \qquad\qquad \textbf{loop}\ \textbf{or}_i\ t_i \Longrightarrow \mathit{spec}\,S_i) \\
\mathit{spec}(\textbf{var}\ t|t_i\ \textbf{in}\ S) & = & (\textbf{var}\ t|t_i\ \textbf{in}\ \ \mathit{spec}\,S) \\
\mathit{spec}(\textbf{con}\ e\ \textbf{in}\ S) & = & (\textbf{con}\ e\ \textbf{in}\ \ \mathit{spec}\,S) \\
\mathit{spec}(\textbf{rec}\ S_1\ \textbf{vary}\ t\ \textbf{in}\ S_2) & = & S_1 \\
\mathit{spec}\,\underline{I} & = & \underline{I} \\
\mathit{spec}(\textbf{proc}\ S\ \textbf{in}\ T) & = & (\textbf{proc}\ \ \mathit{spec}\,S\ \textbf{in}\ \ \mathit{spec}\,T) \\
\mathit{spec}([\ a_1, \ldots, a_n\ ]\ S) & = & [\ a_1, \ldots, a_n\ ]\ (\mathit{spec}\,S) \\
\mathit{spec}(S\ [\ t_1, \ldots t_n\ ]) & = & (\mathit{spec}\,S)\ [\ t_1, \ldots t_n\ ] \\
\mathit{spec}(S\ \textbf{ref}\ T) & = & \mathit{spec}\,S
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{imp}\,A & = & A \quad \textbf{where} \quad A \text{ is atomic} \\
\mathit{imp}(S;T) & = & \mathit{imp}\,S; \mathit{imp}\,T \\
\mathit{imp}(\textbf{pre}\ t\ \textbf{in}\ S) & = & (\textbf{pre}\ t\ \textbf{in}\ \ \mathit{imp}\,S) \\
\mathit{imp}(\textbf{if}\ \textbf{or}_i\ t_i \Longrightarrow S_i) & = & (\textbf{if}\ \textbf{or}_i\ t_i \Longrightarrow \mathit{imp}\,S_i) \\
\mathit{imp}\,(\textbf{while}\ t_I\ \textbf{vary}\ t_V & = & (\textbf{while}\ t_I\ \textbf{vary}\ t_V \\
\qquad\qquad \textbf{loop}\ \textbf{or}_i\ t_i \Longrightarrow S_i) & & \qquad\qquad \textbf{loop}\ \textbf{or}_i\ t_i \Longrightarrow \mathit{imp}\,S_i) \\
\mathit{imp}(\textbf{var}\ t|t_i\ \textbf{in}\ S) & = & (\textbf{var}\ t|t_i\ \textbf{in}\ \ \mathit{imp}\,S) \\
\mathit{imp}(\textbf{con}\ e\ \textbf{in}\ S) & = & (\textbf{con}\ e\ \textbf{in}\ \ \mathit{imp}\,S) \\
\mathit{imp}(\textbf{rec}\ S_1\ \textbf{vary}\ t\ \textbf{in}\ S_2) & = & (\textbf{rec}\ S_1\ \textbf{vary}\ t\ \textbf{in}\ \ \mathit{imp}\,S_2) \\
\mathit{imp}\,\underline{I} & = & \underline{I} \\
\mathit{imp}(\textbf{proc}\ S\ \textbf{in}\ T) & = & (\textbf{proc}\ \ \mathit{imp}\,S\ \textbf{in}\ \ \mathit{imp}\,T) \\
\mathit{imp}([\ a_1, \ldots, a_n\ ]\ S) & = & [\ a_1, \ldots, a_n\ ]\ (\mathit{imp}\,S) \\
\mathit{imp}(S\ [\ t_1, \ldots t_n\ ]) & = & (\mathit{imp}\,S)\ [\ t_1, \ldots t_n\ ] \\
\mathit{imp}(S\ \textbf{ref}\ T) & = & \mathit{imp}\,T
\end{array}
$$

Figure 12.2: Specification and Implementation

$$
\begin{array}{lcl}
\mathit{stmt_ok}_{\rho,\alpha}(S_1\ \textbf{ref}\ S_2) & = & \mathit{stmt_ok}_{\rho,\alpha}\,S_1 \wedge \mathit{stmt_ok}_{\rho,\alpha}\,S_1 \wedge \\
& & \qquad \mathit{stmt_fprm_typ}_{\rho}\,S_1 = \mathit{stmt_fprm_typ}_{\rho}\,S_2 \\[1em]
\mathit{stmt_cc}_{\rho}\,p\ (S_1\ \textbf{ref}\ S_2) & = & \mathit{stmt_cc}_{\rho}\,p\ S_1 \wedge \mathit{stmt_cc}_{\rho}\,p\ S_2 \wedge (p \Rightarrow \mathit{imp}\,S_1 \sqsubseteq \mathit{spec}\,S_2) \\[1em]
\mathit{fcntx}_{\rho}\,p\ (S_1\ \textbf{ref}\ S_2) & = & \mathit{fcntx}_{\rho}\,p\ (\mathit{imp}\,S_1)
\end{array}
$$

Figure 12.3: Refinement Well-formation and Correctness

Following Context The context following a **ref** statement is provided by the left hand side of the refinement (i.e. the specification). This is not the strongest context – that would be provided by the implementation – but it greatly simplifies the context assertions and conforms to the top-down development approach.

Nested Refinement The refinement correctness condition for S **ref** T requires the *implementation* of the left hand side S to be refined by the *specification* of the right hand side T. This provides the treatment of nested refinement described shown in Figure 12.1. Let $L = (A_1 \textbf{ ref } A_2); A_3$ and $R = (A_4 \textbf{ ref } (A_6; A_7)); A_5$, using the definitions of *stmt_cc*, *fcntx*, *spec* and *imp* to expand $stmt_cc_\rho\ p\ E$, we obtain:

$$
\begin{aligned}
& stmt_cc_\rho\, p\ E \\
& \quad = stmt_cc_\rho\, p\ (A_0 \textbf{ ref } (L \textbf{ ref } R)) \\
& \quad = stmt_cc_\rho\, p\ A_0 \wedge (p \Rightarrow A_0 \sqsubseteq (A_1; A_3)) \wedge stmt_cc_\rho\, p\ (L \textbf{ ref } R) \\
& stmt_cc_\rho\, p\ (L \textbf{ ref } R) \\
& \quad = stmt_cc_\rho\, p\ L \wedge (p \Rightarrow (A_2; A_3) \sqsubseteq (A_4; A_5)) \wedge stmt_cc_\rho\, p\ R \\
& stmt_cc_\rho\, p\ L \\
& \quad = stmt_cc_\rho\, p\ ((A_1 \textbf{ ref } A_2); A_3) \\
& \quad = stmt_cc_\rho\, p\ A_1 \wedge (p \Rightarrow A_1 \sqsubseteq A_2) \wedge stmt_cc_\rho\, p\ A_2 \wedge stmt_cc_\rho(sp_\rho\, A_1\ p) A_3 \\
& stmt_cc_\rho\, p\ R \\
& \quad = stmt_cc_\rho\, p\ ((A_4 \textbf{ ref } (A_6; A_7)); A_5) \\
& \quad = stmt_cc_\rho\, p\ A_4 \wedge (p \Rightarrow A_4 \sqsubseteq (A_6; A_7)) \wedge stmt_cc_\rho\, p\ (A_6; A_7) \wedge \\
& \qquad\quad stmt_cc_\rho(sp_\rho\, A_4\ p)\ A_5
\end{aligned}
$$

The four instances of $\sqsubseteq$ correspond to the horizontal lines in Figure 12.1.

12.2 Revisiting the Recursion Statement

The treatment of the recursion statement in Chapter 11 can be clarified using the refinement statement.

Statement in a Recursion In the statement (**rec** S **vary** t_v **in** S'), the statement S is the specification of S' and does not reference the recursive variable or the variant constant. Therefore, if S' references the recursive variable (if not, the recursion can be discarded), it has the form S^{+2} **ref** T for some T, which makes a recursive call.

Checking the Proviso of the Recursion Equivalence Using this observation, we use the equivalent statement for recursion defined in Section 11.9 and the definition of

the function $stmt_cc_\rho$ to argue that the proviso of the equivalence is checked as part of the well-formation checking of a recursion statement. Let $u = type_inf_\rho\, t_v$, then

$$
\begin{array}{ll}
stmt_cc_\rho\, \mathbf{t}(\mathbf{rec}\ S\ \mathbf{vary}\ t_v\ \mathbf{in}\ (S^{+2}\ \mathbf{ref}\ T)) & \\
\quad = stmt_cc_\rho\, \mathbf{t}\ (\mathbf{con}\ t_v \mathbf{proc}\ \ldots\ \mathbf{in}\ (S^{+2}\ \mathbf{ref}\ T)) & [\text{expanding the equivalence}] \\
\quad \Rightarrow (\forall u \cdot stmt_cc_{\rho+u}(\underline{0} = t_v^+)\ (\mathbf{proc}\ \ldots\ \mathbf{in}\ (S^{+2}\ \mathbf{ref}\ T))) & [stmt_cc \text{ for } \mathbf{con}\ ] \\
\quad \Rightarrow (\forall u \cdot stmt_cc_{\rho+u}(\underline{0} = t_v^+) & [\text{definition of } \mathbf{proc}\ ] \\
\qquad\qquad (S^{+2}\ \mathbf{ref}\ (T[0 := \{0 \leq t_v^{+2} \wedge t_v^{+2} < \underline{1}\}; S^{+2}])^-)) & \\
\quad \Rightarrow (\forall u \cdot (\underline{0} = t_v^+) \Rightarrow & [stmt_cc \text{ for } \mathbf{ref}] \\
\qquad\qquad S^+ \sqsubseteq (T[0 := \{0 \leq t_v^{+2} \wedge t_v^{+2} < \underline{1}\}; S^{+2}])^-) & \\
\quad \Rightarrow S \sqsubseteq ((T[0 := \{0 \leq t_v^{+2} \wedge t_v^{+2} < \underline{1}\}; S^{+2}])^-)[0 := t_v^+])^- & [\text{substitution}] \\
\quad \Rightarrow \textit{the recursion proviso} &
\end{array}
$$

A case of $stmt_cc_\rho$ for recursion has already been given (see Section 11.3). A similar argument can be used to derive this from the recursion equivalence statement.

12.3 Refinement Texts

The overall development of a program by refinement from a specification is called a *refinement text*.

Abstract Syntax A refinement text belongs to the syntactic category $\mathcal{U}$ of units, introduced in Section 8.3.

$$\mathcal{U}_x = \ldots \mid \mathbf{reftext}\mathcal{I}\ \mathbf{seq}\mathcal{I}\ \mathcal{S}_x \qquad \mathcal{U}_i = \ldots \mid \mathbf{reftext}\,\mathcal{S}_i$$

A refinement text has a name and a list of directly imported theories, in a similar way to a theory. The program development is represented by a single statement, allowing a text to be written without any refinement steps, if required.

Well-Formation The visibility rules of the refinement text – specified by a definition of the encoding function $[\![\mathbf{reftext}\, l\ ls\ s]\!]_\iota$ – are similar to those for a theory (see Section 8.3). A refinement text is well-formed when the statement component is well-formed. A refinement text does not define any new constants and therefore may not be imported by another unit declaration. Cases of the functions $unit_ok_\kappa$ and $unit_type_\kappa$, which would complete the formal definition, are similar to the corresponding ones for theories and are omitted.

12.4 Validity of Checking Refinements

The following theorem expresses the validity of checking a refinement text.

Theorem 12.1 Stepwise Development *For any statement S and environment ρ such that S is well-formed (i.e. $stmt_ok_{\rho,\alpha}\, S$ for some α), then:*

$$stmt_cc_\rho\, p\ S \Rightarrow stmt_cc_\rho\, p\ (spec\, S) \wedge stmt_cc_\rho\, p\ (imp\, S) \wedge (p \Rightarrow spec\, S \sqsubseteq imp\, S)$$

for any context predicate p.

Note that the reverse implication does not hold, since a development could contain an incorrect step, but still be correct overall. A simple example of this is:

$S \sqsubseteq$ **abort**	incorrect
abort $\sqsubseteq S$	correct

The validity of this theorem primarily depends on:

(i) mononoticity of all the statement constructors, so that any component statement can be replaced by a refinement, and

(ii) validity of the rules for propagating context assertions, so that context p is propagated to a statement S only when S can be refined by $\{p\}S$.

These properties are considered in Chapter 11.

Chapter 13

Conclusions

In this thesis, we have presented a language for program development using the refinement calculus. The well-formation rules and proof semantics of the language have been specified formally and a prototype tool developed from the specification. In this chapter, we summarise the work and suggest directions for further development.

13.1 Summary of Achievements

The feasibility of using a universal law for program refinement has been demonstrated. Compared to transformational refinement using a theorem prover, refinement using a universal law is a smoother development of existing programming methods. The programmer can use operational intuition – *how does this statement behave* – to replace one statement by another with allowed behaviour. The refinement laws add to the programmer intuition but are not used explicitly. The size of refinement steps can be varied, avoiding complex derivations of obvious development steps. Context is propagated automatically, allowing Back's generalisation of the familiar assignment statement to be used. Morgan's specification statement, which is the alternative generalisation of assignment, is appropriate when the context needs to be captured explicitly at each step but the connection to the basic assignment is less clear.

Data refinement is not (yet) supported, but a simulation of data refinement in one of the case studies shows how this could be achieved in a way consistent with our overall approach.

The use of a simple predicate simplifier has been shown to be sufficient to allow the proof obligations – the lemmas of the correctness argument of which the refinement derivation is the main part – to be reduced in both number and complexity. Review of proof obligations is a viable verification method, provided the refinement steps are small. Indeed, when the refinement steps correspond to Morgan's refinement laws, the proof obligations become trivial or disappear completely, to (almost) the same extent

achieved by using the laws explicitly.

The B-Tool/AMN is the most similar existing approach. One feature distinguishing our approach is that refinements can be of any size. Although the theory underlying the B-Tool/AMN is equivalent to the theory of predicate transformers on which our approach is based, B-Tool/AMN allows refinement only between procedures.

The use of subtypes and dependent types is another feature distinguishing our work from the B-Tool/AMN and from Morgan's refinement calculus, which both, in different ways, capture types as invariants. Subtypes and dependent types are integrated smoothly into the programming language, taking account of the context created by the program statements.

Subtypes are shown to provide an accurate model of the runtime constraints of a programming language. Operators with (dependent) subtypes can be used to model integers of limited range. Since these operators can be introduced as the final stage of refinement, the proof obligations are similar to those which arise using Morgan's alternative approach.

The requirement that all variables have an explicit type – a standard practice in imperative programming languages – allows a simple type inference algorithm to be used. Polymorphism is achieved by allowing types to be arguments to functions; this approach is made convenient by enabling the type checker to infer these types in the majority of cases.

13.2 Further Work

The work described in this thesis is intended to provide a foundation for a flexible approach to the validation of a program refinement argument, with formal proof as only one of a number of options. However, this potential has not yet been developed. Applicable verification techniques are testing, including the generation of both test cases [DF93, SC96] and test oracles [PP94], model checking [Jac94] and even runtime checking. Testing could be applied before refinement is complete using animation [HST97].

Further work is also needed to extend the language and to formalise the notion of 'code'. The treatment of language features such as functions, modules and abstract datatypes in languages supporting proof is established [Hor79] and can be adapted to our framework. Other features, such as procedure parameters and exceptions [KM95], also appear feasible. Type parameters and dependent typing so far have only a small role in the language of statements and possible extensions should be explored.

A subset of the language is 'code'. Although we have shown how code features can be modelled using subtypes, we have not considered how code subsets are defined or translated to a mainstream language.

13.3 Refinement in the Mainstream

Despite the growing success of the B-Tool/AMN, formal methods in general and program refinement in particular are not yet mainstream development techniques[1]. Some have argued [Hoa86] that formal methods are necessary to achieve dependable software[2]. To date, this has not been borne out by experience; for example Rushby, examining the application of formal methods to the certification of systems in civil avionics [Rus93, p 135], reports that

> *Overall, present development and assurance practices for aircraft seem to have worked so far: to my knowledge, there is no case of a software fault leading to a serious failure condition in a military or commercial airplane.*

However, if achieving dependable software is possible, it is also expensive: in the context of testing to RTCA/DO-178B, Rushby reports [Rus93, p 146] *'I have heard estimates that up to 30% of all software development costs may be consumed in testing ...'*. This suggests an alternative vision for the use of formal methods: reducing the cost of achieving highly dependable software.

We believe that this vision can be realised only by the exploitation of theories such as the refinement calculus in combination with the best existing practices and tools. The use of formal methods should be incremental, enhancing existing practices and allowing flexibility in matters such as the degree of rigour and the order of lifecycle steps. The work described in this thesis is intended to be a small step in this direction.

[1] Parnas discusses the reasons for this in the introduction to a compilation of B-Tool/AMN case studies [SS99].

[2] A decade later, Hoare [Hoa96] examined why his prediction has been shown to be false. Among the reasons given, he notes that software systems can operate reliably despite containing errors.

Appendix A

Syntax Definition

This chapter describes a concrete syntax for the language, noting significant differences between the concrete syntax and the abstract syntax.

Notation The concrete syntax is described in an EBNF using the following notation:

`KEYWORD`	Terminal symbol
`":="`	Terminal symbol
`Category`	Non-terminal
`E F`	E followed by F
`E | F`	E or F
`[ E ]`	Zero or one occurrences of E
`{ E }`	Zero or more occurrences of E

A.1 Lexical Rules

A.1.1 Lexical Tokens

The forms of lexical tokens are alphanumeric, numbers, non-alphanumeric, brackets, strings and character literals. Both alphanumeric and non-alphanumeric `OpTok` tokens can be used either as *operators* or as *identifiers*, as explained in Section A.1.3 below.

```
AlphaTok  ::= Letter { Letter | OpAChar | Digit }
NumTok    ::= Digit { Digit }
BrackTok  ::= "(" | ")"
OpTok     ::= OpChar { OpChar | OpAChar }
CharLTok  ::= "'" ( StringChar | """ ) "'"
StringTok ::= """ { StringChar | "'" } """

OpAChar    ::= "_"
```

```
OpChar     ::= "{" | "}" | "[" | "]" | "<" | ">" | "\" | "/"
             |  "." | "," | ";" | ":" | "&" | "?" | "!"
             |  "$" | "%" | "^" | "*" | "+" | "-" | "="
             |  "~" | "#" | "@" | "|" | "`"
StringChar ::= Letter | Digit | OpAChar | "(" | ")" | SpaceChar
```

The £ character is also in `OpChar`.

A.1.2 Reserved Words

Some tokens belonging to both `AlphaTok` and `OpTok` are reserved words. The reserved words from `AlphaTok` are case sensitive: thus `END` is a reserved word, while `end` and `eNd` are not[1]. The reserved words are:

ABORT	ANY	AXIOM	BEGIN	BINDER
BY	CASE	CHOOSE	CON	ELSE
END	EXISTS	FORALL	IF	IN
INFIX	INFIXL	INFIXR	IS	LEFT
LET	LOOP	MEAS	OF	OR
PRE	PREC	PREFIX	POSTFIX	PROC
REC	REF	RES	RIGHT	
SEE	SKIP	SPEC	THEN	THEOREM
THEORY	TYPE	VAL	VALRES	VAR
VARY	WHERE	WHILE		

The reserved words from `OpTok` are:

```
{    }    [  ]   @   ;    :    :=   ::=   <<
>>   ==>  \  ->  =   /=   ==   |    "     '
```

Note that the comma is not reserved since it is declared as the operator for creating pairs (see page 33), although it is treated as reserved in the syntax of statements.

A.1.3 Operator Declarations

An operator declaration `Opdecl` is required to make a token behave as an operator; both forms of identifier may be used as operators.

```
Opdecl    ::= Fixity Op { " " Op }
Fixity    ::= INFIX Digit | INFIXL Digit | INFIXR Digit
            | PREFIX | POSTFIX | BINDER | LEFT | RIGHT
Op        ::= AlphaTok | OpTok
```

Tokens from `AlphaTok` and `OpTok` which are not declared as operators are identifiers `Id`.

[1]In this respect reserved words are treated in the same way as identifiers. This is appropriate to avoid having to distinguish pre-declared identified from reserved words.

A.2 Texts, Theories and Declarations

A text is made up of theories and a refinement text. A theory is a named collection of declarations.

```
Text   ::= { Theory } RefText
Theory ::= [ SEE Id { Id } ]
           THEORY Id IS
             [ Opdecl { ";" Opdecl } ";" ]
             Decl { ";" Decl }
           END
```

The keyword SEE is followed by a list of directly visible theories. Other theories which precede the theory being declared are indirectly visible. The category Decl includes declarations, definitions or datatypes. Identifiers introduced may be operators (Op).

```
Decl ::= AXIOM Id { Arg } "==" Term
       | THEOREM Id { Arg } "==" Term
       | IdOp [ Defs ] ":" Term
       | IdOp [ Defs ] { Arg } DefRhs

IdOp ::= Id | "(" Op ")"
Defs ::= "[" Int "]"
Arg  ::= "(" Id : Term ")"

Case  ::= Id [ { Arg } Id ]
```

A.2.1 Definitions

The syntax for the right hand side of a definition is:

```
DefRhs ::= [ ":" Term ] "==" Term
         | CASE Arg [ ":" Term ] "==" Alterns END
         | PREC Arg ":" Term == Alterns END
         | REC { Arg } DefRec
         | "::=" Case { "|" Case }

DefRec ::= ":" Term  "==" Term MEAS Term END
         | CASE Arg ":" Term  "==" Alterns MEAS Term END
```

The syntax is elaborated to allow functions to be declared without explicit use of λ-abstractions and the related forms for primitive and general recursion. These can be combined with definition by cases. However, the syntax is still more elaborate than the 'pattern matching' syntax used in, for example, Haskell.

A.3 Refinement Text

Refinement labels are used to allow the refinement text to be written in a linearised form. The complete text of a refinement starts with a specifying statement and is followed by a list of refinements.

```
RefText ::= [ SEE Id { Id } ]
            SPEC Id IS Stmt END
            { Refinement }

Refinement ::= REF Label BY Stmt END
```

A.4 Statements

The following precedence rules apply to statements: actual parameters (on an application statement) have highest precedence, labels are next and sequential composition has lowest precedence.

```
Stmt        ::= Label-Stmt [ ";" Stmt ]
Label-Stmt  ::= [ "<<" Label ">>" ] Applic-Stmt
Applic-Stmt ::= Basic-Stmt [ Actuals ]
```

A.4.1 Basic Statements

The syntactic category `Basic-Stmt` include all other statements – no precedence rules are required since the composite statements are 'bracketed'. The `BEGIN ... END` block acts as a pair of brackets around a statement; it is needed in order to be able to label the sequential composition of two statements.

```
Basic-Stmt ::= SKIP
             | ABORT
             | PRE Term IN Stmt END PRE
             | BEGIN Stmt END
             | VAR Vars IN Stmt END VAR
             | CON Id "==" Term IN Stmt END CON
             | IF Guarded-Stmts END IF
             | WHILE Term VARY Term LOOP Guarded-Stmts END LOOP
             | REC Id IS Abstract-Stmt VARY Term END REC
             | PROC Id IS Abstract-Stmt IN Stmt END PROC
             | Asgn-or-Call-Stmt
```

A.4.2 Formal and Actual Parameters

An abstraction statement `Abstract-Stmt` may have formal parameters. A label cannot appear on the left hand side of the formal parameters in an abstraction statement,

although the statement within the abstraction may be labelled. This is consistent with the restriction given in Section 11.7.4.

```
Abstract-Stmt ::= [ Formals ] Stmt
```

Lists of actual parameters are separated by commas. Within an abstraction, the substitution mechanism may apply to more than one formal parameter. A result parameter may have an initial value: the syntax is the same as a variable declaration.

```
Actuals ::= "[" Term { "," Term } "]"
Formals ::= "[" Formal { ";" Formal } "]"
Formal  ::= Init-Formal | Non-Init-Formal
Non-Init-Formal ::= ( VAL | VALRES | REF )
                      Id ":" Term { ";" Id ":" Term }
Init-Formal     ::= RES Vars
```

A.4.3 Assignment and Call

Both an assignment and a procedure call start with an identifier. The syntax can be disambiguated by noting that the symbol following the identifier in a procedure call must one of `[ ; END VARY IN`.

```
Asgn-or-Call-Stmt ::= Left [ ":=" Right ]

Right             ::= [ ANY Id { "," Id } ] WHERE Term
                   |  Term { "," Term }
```

Left Hand Side of an Assignment The syntax of the left hand side `Left` shows the forms of term which can be used on the left hand side of an assignment.

```
Left ::= PrimLeft { LeftArg }
      |  IdDot { "[" Term "]" } PrimLeft
      |  Id "[" Term "]"  { "[" Term "]" } PrimLeft

LeftArg ::= "^" PrimTerm |  PrimTerm

PrimLeft ::= Id |  "(" Left ")"
```

Note that:

(i) In a left value, the map index operator `^` parses like function application, which is left associative and has higher precedence than any binary operator. As a result the term `m^a+b^c` does not parse as a left value, but both `m^(a+b)^c` and `m^(a+b^c)` do.

(ii) a left value formed from a directly-visible datatype accessor parses as a function application (unless if has explicit default parameters). For example, in `f a` the identifier `f` could be a variable or a datatype accessor.

(iii) There is an ambiguity between a procedure call and a data type accessor with an explicit default parameter. The former takes precedence.

A.4.4 Variable Declaration and Initialisation

A `VAR .. END VAR` block contains a sequence of variable declarations, separated by semicolons. Variables may be initialised.

```
Vars ::= Id ":" Term [ ":=" Init-Value ] { ";" Vars }

Init-Value ::= WHERE Term | Term
```

In the abstract syntax all variable declarations include an initial value term. In the concrete syntax, a form is provided to specify the initial value by a predicate. When an explicit initialisation is omitted, the variable is initialised to an arbitrary value *within its type*.

A.4.5 Guarded Statement

The body of the choice and loop is a disjunction of statements, each guarded by a predicate. A label may not appear on the guard.

```
Guarded-stmts ::= Guarded-stmt { OR Guarded-stmt }
Guarded-stmt  ::= Pred "==>" Stmt
```

A.5 Terms

The syntax of terms includes the function type, the binary operator term, the unary operator term and applications. The function type constructor associates to the right. Binary operator terms are parsed in accordance with their declared precedence and associativity. All unary operators have higher precedence than binary operators; a postfix unary operator binds tighter than a prefix unary operator.

```
Term    ::= [ Id ":" ] BinTerm "->" Term | BinTerm
BinTerm ::= PreTerm | BinTerm BinOp BinTerm
PreTerm ::= UnyTerm | PreOp PreTerm
UnyTerm ::= AnnTerm | UnyTerm PostOp
```

The equality operators (precedence 4, no associativity) are built-in.

```
BinOp  ::= "=" | "/=" | binary operators
PreOp  ::= prefix operators
PostOp ::= postfix operators
```

Function applications bind tightest and associate to the left in the standard manner. Implicit parameters can be made explicit in an application, using the method described in Section 3.2.3. An annotation binds less strongly than an application so that in a type annotation `t : T`, the type `T` may be an application or a primitive term.

```
AnnTerm ::= AppTerm "[" ":" AppTerm "]"
AppTerm  ::= AppTerm AppTermI |  AppTermI
AppTermI ::= AppTermI "[" Term "]" |  PrimTerm
```

The syntactic category `PrimTerm` includes all the primitive terms and the composite terms with bracketed syntax.

```
PrimTerm ::= "TYPE"
           | IdOp
           | Id "." IdOp
           | NumTok
           | CharTok
           | StringTok
           | "(" BrackTerm ")"
           | "{" SubtTerm "}"
           | PREC [Id ":"] Term "->" Term IS Alterns END
           | CASE Term OF Alterns END
           | REC Id { Arg } ":" Term IS Term MEAS Term END
           | LET LetDef { ";" LetDef } IN Term END
           | IF Term THEN Term ELSE Term END
           | LDLimOp Term { ; Term} RDLimOp

BVar      ::= Id ":" Term
BrackTerm ::= BinderOp BVar { ";" BVar } "@" Term
           |  Term
LetDef    ::= BVar "==" Term
SubtTerm  ::= BVar "|" Term
           |  Term
```

The λ-abstraction operator \ and the quantifiers act as binders, together with any declared binders.

```
BinderOp ::= "\" | FORALL | EXISTS | CHOOSE | binder operators
```

Alternatives are used in both case terms and primitive recursion. The left hand side of an alternative is a pattern, which is a sequence of identifiers.

```
Alterns ::=  Altern { "|" Altern }
Altern  ::=  Pattern "==" Term
Pattern ::=  { Id }
```

A.6 Alternative Syntax for Refinements

A specification and its refinements form a tree. In the syntax using labels, the tree is flattened and the labels show the hierarchy. An alternative approach is to represent the

```
@ The specification is given using a non-deterministic assignment.
<<between.ref>>=
  VAR x : Int IN
    PRE <<provided x large enough>> IN
      <<assign new value between one and x>>
    END PRE
  END VAR
@ %def x

@ The operation is impossible unless [[x]] is initially at least 3.
<<provided x large enough>>=
  3 <= x

@ This is the non-deterministic assignment. It is refined by a
procedure call.
<<assign new value between one and x>>=
  x := ANY nx WHERE 1 < nx && nx < x
REFINED_BY
  PROC BetW IS [VAL a : Int; VALRES b : Int]
    <<between procedure>>
  IN
    BetW[1, x]
  END PROC
```

Figure A.1: Input Text to Noweb

---

tree directly using nesting and a new reserved word REFINED_BY. For example:

```
VAR x : Int IN
  PRE 3 <= x IN
    x := ANY nx WHERE 1 < nx && nx < x
  REFINED_BY
    PROC BetW IS [VAL a : Int; VALRES b : Int]
      PRE a <= b IN
        b := ANY x WHERE a <= x && x <= b
      REFINED_BY
        b := (a + b) / 2
      END PRE
    IN
      BetW[1, x]
    END PROC
  END PRE
END VAR
```

However, this is too unwieldy to be used directly. But it can be constructed using a Web tool, such as Noweb [Ram94]. Figure A.1 shows the Noweb source for part of the simple refinement given above, while Figure A.2 shows the LaTeX output. HTML can also be produced. The text shown above is a reformatted version of the "tangled" output of Noweb.

The specification is given using a non-deterministic assignment.

212a ⟨*between.ref* 212a⟩≡
```
VAR x : Int IN
  PRE ⟨provided x large enough 212b⟩ IN
    ⟨assign new value between one and x 212c⟩
  END PRE
END VAR
```
Defines:
  x, never used.
Root chunk (not used in this document).

The operation is impossible unless **x** is initial at least 3.

212b ⟨*provided x large enough* 212b⟩≡
```
3 <= x
```

This code is used in chunk 212a.

This is the non-deterministic assignment. It is refined by a procedure call.

212c ⟨*assign new value between one and x* 212c⟩≡
```
  x := ANY nx WHERE 1 < nx && nx < x
REFINED_BY
  PROC BetW IS [VAL a : Int ; VALRES b : Int]
    ⟨between procedure⟩
  IN
    BetW[1, x]
  END PROC
```
This code is used in chunk 212c.

Figure A.2: LaTeXproduced by Noweb

Appendix B

Standard Declarations

In this chapter a small library of standard declarations[1] is given. The declarations illustrate how a library of standard types and functions could be prepared. Such a library is required to make the primitive language of terms a practical specification language; examples include the Z-toolkit [Spi89] and the theories which form part of the PVS thoerem prover [OSR93].

B.1 Functions

Function override and composition are defined. Predicates which characterise functions are also useful.

```
> SEE Bool Pair
> THEORY Function IS
>   INFIXL  1 <+> o ;

>   override [2] (A:TYPE) (B:TYPE) (f:A->B) (a:A) (b:B) ==
>     (\x:A @ IF x = a THEN b ELSE f x END) ;

>   (<+>) [2] (A:TYPE) (B:TYPE) (f:A->B) (p:A#B) ==
>     override f (fst p) (snd p) ;

>   (o) [3] (A:TYPE) (B:TYPE) (C:TYPE) (f:B->C) (g:A->B) (a:A) ==
>     f (g a) ;

>   injective [2] (A:TYPE) (B:TYPE) (f:A->B) ==
>     (FORALL a:A; b:A @ f a = f b => a = b) ;

>   surjective [2] (A:TYPE) (B:TYPE) (f:A->B) ==
>     (FORALL b:B @ (EXISTS a:A @ f a = b)) ;
```

[1]In addition to the theories given in Chapter 3.

```
>   bijective [2] (A:TYPE) (B:TYPE) (f:A->B) ==
>     injective f && surjective f
> END
```

This theory is obviously incomplete: many other standard definitions could be added.

B.2 Sets

Sets are modelled as boolean functions.

```
> SEE Bool
> THEORY Set IS
>   INFIX  4 :: ;  INFIXL 3 /\ ;
>   INFIXL 2 \/ ;  INFIX  1 :< -- ;

>   Set (A:TYPE) == A -> Bool ;

>   emptySet [1] (A:TYPE) (a:A) == False ;
>   singleSet [1] (A:TYPE) (a:A) (x:A) == x = a ;
>   isemptySet [1] (A:TYPE) (s:Set A) == (FORALL a:A @ not (s a)) ;
>   (\/) [1] (A:TYPE) (s:Set A) (t:Set A) (a:A) == s a || t a ;
>   (/\) [1] (A:TYPE) (s:Set A) (t:Set A) (a:A) == s a && t a ;
>   (--) [1] (A:TYPE) (s:Set A) (t:Set A) (a:A) == s a && not (t a) ;
>   (::) [1] (A:TYPE) (a:A) (s:Set A) == s a ;
>   (:<) [1] (A:TYPE) (s:Set A) (t:Set A) == (FORALL a:A @ s a => t a)
> END
```

B.3 Map

A map, or partial function, is modelled by a dependent pair: the first element of the pair is the set of values in the domain of the partial function and the second element is a function from the subtype characterised by the set.

```
> SEE Bool Pair Function Set
> THEORY Map IS
>   INFIXL 5 ^ ; INFIXR 5 -+> ;  INFIXL 1 [+] ;

>   Map (A:TYPE) (B:TYPE) == (PAIR d:Set A @ {d} -> B) ;
>   (-+>) == Map ;

>   emptyMap [2] (A:TYPE) (B:TYPE) : Map A B ==
>     (pair d:Set A @ {d} -> B)
>       emptySet
>       (\a: {emptySet [A] } @ (CHOOSE b:B @ True)) ;

>   elemMap [2] (A:TYPE) (B:TYPE) (m:Map A B)
```

```
>     (i: {fst m}) == snd m i ;

>   (^) [2] (A:TYPE) (B:TYPE) == elemMap [A] [B] ;

>   dmnMap [2] (A:TYPE) (B:TYPE) (m:Map A B) == fst m ;

>   updateMap [2] (A:TYPE) (B:TYPE) (m:Map A B)
>                         (a: {dmnMap m}) (b:B) : Map A B ==
>     (pair d:Set A @ {d} -> B)
>       (dmnMap m)
>       ((snd m) <+> (a,b)) ;

>   ([+]) [2] (A:TYPE) (B:TYPE) (m:Map A B) (p: {dmnMap m} # B) ==
>     updateMap m (fst p) (snd p) ;

>   addMap [2] (A:TYPE) (B:TYPE) (m:Map A B) (a:A) (b:B) : Map A B ==
>     (pair d:Set A @ {d} -> B)
>       (dmnMap m \/ singleSet a)
>       (\aa: {dmnMap m \/ singleSet a} @
>         IF aa = a THEN b ELSE m^aa END) ;

>   deleteMap [2] (A:TYPE) (B:TYPE) (m:Map A B) (a:A) : Map A B ==
>     (pair d:Set A @ {d} -> B)
>       (dmnMap m -- singleSet a)
>       (\aa: {dmnMap m -- singleSet a} @ m^aa)
> END
```

A map can be updated in an assignment statement (see Sections 11.1 and 11.6.2).

B.4 Arrays and the `Upto` Subtype

Finite subtypes of integers can be constructed using `..` and `upto`.

```
> SEE Bool Int
> THEORY Upto IS
>   INFIX  1 .. to ;

>   (to) (l:Int) (u:Int) == (\x:Int @ l <= x && x <= u) ;
>   (..) (l:Int) (u:Int) == {l to u} ;
>   upto (size:Int) == 0 .. (size - 1)
> END
```

An array type constructor is introduced. For simplicity, the first index of all arrays is 0.

```
> SEE Bool Int Upto
> THEORY Array IS
>   Array (size:Nat) (T:TYPE) == (upto size) -> T
> END
```

An array can be updated in an assignment statement (see Section 11.6.2).

B.5 Finiteness and Cardinality

Finite Sets Finite sets have a defined size.

```
> SEE Bool Int Function Set Upto
> THEORY Finite IS
>   finite [1] (A:TYPE) (s:Set A) ==
>     (EXISTS n:Nat; f: upto n -> {s} @ bijective f) ;
>   FSet (A:TYPE) == {s:Set A | finite s} ;
>   size [1] (A:TYPE) (s:FSet A) ==
>     (CHOOSE n:Nat @ (EXISTS f: upto n -> {s} @ bijective f))
> END
```

Finite Map An important subtype of the map type is the finite map.

```
> SEE Bool Map Finite
> THEORY FMap IS
>   finite_map [2] (A:TYPE) (B:TYPE) (m:Map A B) == finite (dmnMap m) ;
>   FMap (A:TYPE) (B:TYPE) == {finite_map [A] [B] } ;

>   THEOREM empty (A:TYPE) (B:TYPE) == finite_map (emptyMap [A] [B])
> END
```

B.6 Collections and Pointers

Pointers can be introduced, using explicit collections. Collections were introduced for the Stanford Pascal Verifier [LS79] (where they are called *reference classes*) and in the language Euclid [LGH+78].

```
> SEE Bool Pair Set Map FMap
> THEORY Collection IS

>   Ptr : TYPE ; Nil : Ptr ;
>   AXIOM NPtr == (EXISTS p:Ptr @ p /= Nil) ;
>   NPtr == {p:Ptr | p /= Nil} : TYPE1 ;
>   Collection (A:TYPE) == FMap NPtr A ;

>   isemptyC [1] (A:TYPE) (c:Collection A) == isemptySet (dmnMap c) ;

>   newCp [1] (A:TYPE) (c:Collection A)  ==
>     (CHOOSE p:NPtr @ not (p :: dmnMap c)) ;

>   newC [1] (A:TYPE) (c:Collection A)
```

```
>           (p: {p:NPtr| not (p :: dmnMap c)}) (a:A) : Collection A ==
>     addMap c p a ;

>   delC [1] (A:TYPE) (c:Collection A) (p:NPtr) : Collection A ==
>     deleteMap c p ;

>   THEOREM newC1 (A:TYPE) ==
>    (FORALL c:Collection A ; a:A ; p:NPtr @
>      p :: dmnMap c => (newC c (newCp c) a)^p = c^p) ;

>   THEOREM newC2 (A:TYPE) ==
>    (FORALL c:Collection A ; a:A @
>      LET p:NPtr == newCp c IN (newC c p a)^p = a END) ;

>   THEOREM delC (A:TYPE) ==
>    (FORALL c:Collection A ; p: {dmnMap c} ; q: {dmnMap c} @
>      p /= q => (delC c p)^q = c^q)
> END
```

Example As a simple example of the use of pointer and collections, a type which could be used to implement a linked list is defined:

```
ListElem == Int16 # Ptr ;
LinkedList == Collection ListElem # Ptr
```

The linked list is a pair consisting of a collection and a pointer which indexes the head of the list.

Pointer Types The definitions above result in a weaker type system than that of Pascal. Here, all pointers are of the same type whereas in Pascal pointers of type $\uparrow T$ point to variables of type T. We can attempt to replicate this by declaring a type constructor: `Ptr : TYPE -> TYPE` . However, this results in an illegal recursion. Pascal depends on the property that a pointer is always the same size, no matter which type is pointed to, allowing a type to be defined in terms of itself. No way of specifying this property within the language of terms has been found.

Storage Allocation A collection is represented by a *finite* map so that it is always possible to choose a new pointer which is not already in the collection. The definitions given above do not model the exhaustion of storage space.

B.7 Lists

A datatype for finite lists is declared.

```
> SEE Bool Int
> THEORY List IS
>   INFIX  3 <-- ;
>   INFIXL 2 ++ ;
>   INFIXL 8 !! ;

>   List [1] (A:TYPE) ::= nil empty |
>                         cons (head:A) (tail:List [A]) nonempty ;

>   length [1] (A:TYPE) == PREC List [A] -> Nat IS
>                            nil      ==> 0 |
>                            cons h t ==> t + 1
>                          END ;

>   (!!) [1] : A:TYPE -> l:List [A] -> {j:Nat | j < length l} -> A ;

>   elems [1] (A:TYPE) (l:List [A]) (s:Int) (f:Int) ==
>     (\a:A @ (EXISTS i: {j:Nat | j < length l} @
>       s <= i && i < f && l !! i = a)) ;

>   map [2] (A:TYPE) (B:TYPE) (f:A->B) ==
>     PREC List [A] -> List [B] IS
>       nil      ==> nil |
>       cons h t ==> cons (f h) t
>     END
> END
```

B.8 Characters and Strings

Characters can be introduced by an enumerated type.

```
THEORY Char IS
  Char ::=
    char_0  | char_1  | char_2  | char_3  | char_4  | ...
END
```

Ideally, the representation of a string must allow the used part of the string to vary in length upto the length allocated. We outline two approaches. Both approaches have an important feature in common: the type system ensures that no string is accessed or updated beyond the allocated space.

String with an Index In this approach, a string is represented by a pair, consisting of an array of characters and a length. Since arrays are indexed from zero, the length is the next array index after the end of the string, which could be beyond the end of the array.

```
SEE Bool Int Array Char
```

```
THEORY String IS
  String (l:Nat) == Array Char l # 0 .. l ;
  string_empty == ((\i:upto 0 @ (CHOOSE c:Char @ True),0) : String 0 ;
  string_len [1] (l:Nat) (s:String l) == snd s ;
  string_elem [1] (l:Nat) (s:String l) (i:upto l) == (fst s) i ;

  string_add [1] (m:Nat) (c:Char) (s:String m) : String (m+1) ==
    ((\i:upto (m+1) @
      IF i = 0 THEN c ELSE string_elem s (i-1) END),
    m + 1) ;

  string_concat [2] (l:Nat) (m:Nat) (s:String l) (t:String m)
                    : String (string_len s + string_len t) ==
    LET sl:Nat == string_len s ; tl:Nat == string_len t IN
      (\i:upto (sl + tl) @
        IF i < sl THEN string_elem s i
                  ELSE string_elem t (i - sl) END), sl + tl
    END
END
```

Strings with NUL Termination An alternative definition of string uses a special character to mark the end of the string.

```
THEORY StringN IS
  StringN (l:Nat) ==
    {s:Array Char (l+1) | (EXISTS i:upto (l+1) @ s i = NUL)} ;

  stringn_len [1] (l:Nat) (s:StringN l) ==
   (CHOOSE i:upto l @ s (i+1) = NUL &&
     (FORALL j:upto (i+1) @ s j /= NUL))
END
```

Character and String Literals Character and string literals are represented in the abstract syntax using the definitions above:

Literal	Equivalence
'a'	a_char
"ab"	string_add a (string_add b empty_string)

Appendix C

Proofs

C.1 Loops

C.1.1 Implementing Loops

To prove Theorem 11.2, we use the standard proof rule for the loop (see [Gri81]). If, for any predicate I and integer valued expression E:

1. $\bigwedge_i I \wedge B_i \Rightarrow wp(S_i, I)$, and
2. $I \wedge \bigvee_i B_i \Rightarrow 0 < E$, and
3. $\bigwedge_i I \wedge B_i \Rightarrow wp(S_i, E < v)[v := E]$

then

$$I \Rightarrow wp(\mathbf{dood}, I \wedge \neg \bigvee_i B_i)$$

Proof of Theorem 11.2, Page 189 For an arbitrary postcondition Q:

$$\begin{array}{ll}
wp(\mathbf{loop}, Q) & \\
\quad = I \wedge & \text{[wp of loop]} \\
\quad\quad \bigwedge_i(\forall \vec{x} \cdot I \wedge B_i \Rightarrow wp(S_i, I \wedge E < v)[v := E]) \wedge & \\
\quad\quad (\forall \vec{x} \cdot I \wedge \bigvee_i B_i \Rightarrow 0 < E) \wedge & \\
\quad\quad (\forall \vec{x} \cdot I \wedge \bigwedge_i \neg B_i \Rightarrow Q) & \\
\quad = I \wedge & [S_i \text{ is conjunctive}] \\
\quad\quad \bigwedge_i(\forall \vec{x} \cdot I \wedge B_i \Rightarrow wp(S_i, I)) \wedge & \\
\quad\quad \bigwedge_i(\forall \vec{x} \cdot I \wedge B_i \Rightarrow wp(S_i, E < v)[v := E]) \wedge & \\
\quad\quad (\forall \vec{x} \cdot I \wedge \bigvee_i B_i \Rightarrow 0 < E) \wedge & \\
\quad\quad (\forall \vec{x} \cdot I \wedge \bigwedge_i \neg B_i \Rightarrow Q) &
\end{array}$$

$$\Rightarrow wp(\textbf{dood}, I \wedge \neg \bigvee_i B_i) \wedge (\forall \vec{x} \cdot I \wedge \bigwedge_i \neg B_i \Rightarrow Q) \qquad \text{[Using the proof rule]}$$

$$\Rightarrow wp(\textbf{dood}, Q) \qquad \text{[de-Morgan, monotonicity]}$$

□

Note that the proof holds even if any of the statements S_i in **loop** are not continuous and there is no integer valued expression E which is a loop variant, since this implies that $wp(\textbf{loop}, Q)$ is false.

C.2 Recursion

C.2.1 Introducing a Recursive Call

Proof of Theorem 11.3, Page 191 For an arbitrary postcondition Q:

$$\begin{array}{l} wp(S_1, Q) \\ \Rightarrow \text{“Since } S_1 \sqsubseteq S_2 \text{ and } S_2 \sqsubseteq (\textbf{con } v = E \textbf{ in } \ldots)\text{”} \\ \quad wp((\textbf{con } v = E \textbf{ in } \ldots), Q) \\ \Rightarrow \text{“Defn. of } wp \text{ for } \textbf{con}\text{”} \\ \quad wp(T_X[X := \{\, 0 \le E < v \,\}; S_2], Q)[v := E] \\ \Leftrightarrow \text{“Using defn. of } wp \text{ for } \textbf{rec}\text{”} \\ \quad wp((\textbf{rec } X \mathrel{\hat{=}} S_2 \textbf{ vary } E \textbf{ in } T_X), Q) \end{array}$$

□

C.2.2 Implementing Recursion

Proof of Theorem 11.4, Page 191 To prove Theorem 11.4 we use the following rule for introducing a recursion, given by Back and von Wright [BW98, §20]:

$$\frac{\{I \wedge t = w\}; S \ \sqsubseteq \ T[X := \{I \wedge t < w\}; S]}{\{I\}; S \ \sqsubseteq \ (\mu X \cdot T)}$$

where I is a boolean term and t is a variant, ranging over some well-founded set (for which $<$ is the well-founded order). Using variable notation (rather than De-Bruijn indices) we write the recursion equivalence as:

$$(\textbf{rec } X \mathrel{\hat{=}} S \textbf{ vary } t \textbf{ in } T) \mathrel{\hat{=}} (\textbf{con } w \mathrel{\hat{=}} t \textbf{ in } (\textbf{proc } X \mathrel{\hat{=}} \{0 \le t < w\}; S \textbf{ in } T))$$

where w is a fresh identifier, and the recursive *proviso* is:

$$S \sqsubseteq (\textbf{con } w \mathrel{\hat{=}} t \textbf{ in } T[X := \{0 \le t < w\}; S])$$

The proviso matches the condition for the introducing a recursion using the rule above, giving:

$$S \sqsubseteq (\mu X \cdot T) \qquad (*)$$

To complete the proof of Theorem 11.4, let R be an arbitrary postcondition:

$wp((\mathbf{rec}\ X \mathrel{\hat{=}} S\ \mathbf{vary}\ t\ \mathbf{in}\ T), R)$
$\Rightarrow$ "Recursion equivalence; wp for **con** and **proc**"
$\quad wp(T[X := \{0 \leq t < w\}; S], R)[t := w]$
$\Rightarrow$ "Using $(*)$ above and monotonicity, since the proviso shows that the variant decreases"
$\quad wp(T[X := \mu X \cdot T], R)$
$\Rightarrow$ "Unfolding of a fix point"
$\quad wp((\mu X \cdot T), R)$

□

Bibliography

[Abr89] J. R. Abrial. A formal approach to large software construction. In de Snepscheut [dS89], pages 1–20.

[Abr91a] J. R. Abrial. Abstract machines. Course notes for BCS FACS B Tutorial, April 1991.

[Abr91b] J. R. Abrial. A refinement case study using the Abstract Machine Notation. In Morris and Shaw [MS91], pages 51–96.

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Amb77] A. L. Ambler. Gypsy: A language for specification and implementation of verifiable programs. *ACM SIGPLAN Notices*, 12(3), March 1977.

[Bac78] R. J. R. Back. On the correctness of refinement steps in program development. Technical Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

[Bac87] R. J. R. Back. Procedural abstraction in the refinement calculus. Technical Report Ser. A, No. 55, Åbo Akademi, Department of Computer Science, Fänriksgatan 3 B, SF-20500, Åbo, Finland, September 1987.

[Bac88] R. J. R. Back. A calculus of refinement for program derivation. *Acta Informatica*, 25:593–624, 1988.

[Bac89] R. J. R. Back. Changing data representation in the refinement calculus. In *Proceedings 22nd Hawaii International Conference of System Sciences*, Kailua-Kona, January 1989.

[Bac91] R. J. R. Back. Refinement diagrams. In *Proceeding 4th Refinement Workshop.* Springer-Verlag, January 1991.

[Bar92] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2 Background: Computational Structures, pages 117–309. Oxford Science Publications, 1992.

[BDM98] P. Behm, P. Desforges, and J.-M Meynadier. METEOR: An industrial success in formal development. In Didier Bert, editor, *B'98: recent advances in the development and use of the B method: second International B Conference, Montpellier, France*, volume 1393 of *Lecture Notes in Computer Science*, pages 26–28, New York, NY, USA, April 1998. Springer-Verlag Inc.

[Beh96] Patrick Behm. Développement formel des logiciels sécuritaires de METEOR. In Henri Habrias, editor, *Proceedings of the 1st Conference on the B method*, Putting into Practice methods and tools for information system design, pages 3–10, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, November 1996. IRIN Institut de recherche en informatique de Nantes.

[BGL+97] Michael Butler, Jim Grundy, Thomas Långbacka, Rimvydas Rukšėnas, and Joakim von Wright. The refinement calculator: Proof support for program refinement. In Lindsay Groves and Steve Reeves, editors, *Formal Methods Pacific'97: Proceedings of FMP'97*, Discrete Mathematics and Theoretical Computer Science, pages 40–61, Wellington, New Zealand, July 1997. Springer-Verlag.

[BH97] J. Barnes and J. Hammond. An independent evaluation of the DRA compliance notation. Technical Report S.P0470.50.1 Issue 2.0, Praxis Critical Systems, March 1997.

[BHS92] R. J. R. Back, J. Hekanaho, and K. Sere. Centipede — a program refinement environment. Reports on Computer Science & Mathematics Ser. A. 139, Åbo Akademi, September 1992.

[BL96] Michael Butler and Thomas Långbacka. Program derivation using the refinement calculator. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference*, volume 1125 of *Lecture Notes in Computer Science*, pages 93–108, Turku, Finland, August 1996. Springer-Verlag.

[BLRvW95] Michael Butler, Thomas Långbacka, Rimvydas Rukšėnas, and Joakim von Wright. Refinement calculator tutorial and manual. April 1995.

[Boo82] H. J. Boom. A weaker precondition for loops. *ACM Transactions on Programming Languages and Systems*, 4(4):668–677, October 1982.

[Bru72] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[But97] Michael Butler. Calculational derivation of pointer algorithms from tree operations. Technical Report DSSE-TR-97-5, Department of Electronics and Computer Science, University of Southampton, December 1997.

[BW89a] R. J. R. Back and J. von Wright. A lattice-theoretical basis for a specification language. In de Snepscheut [dS89], pages 139–156.

[BW89b] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. REX Workshop, Mook, The Netherlands, Springer-Verlag, May 1989.

[BW93] R. J. R. Back and J. von Wright. Statement inversion and strongest postcondition. *Science of Computer Programming*, 20:223–251, 1993.

[BW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, New York, 1986.

[CGM92] B. A. Carré, J. R. Garnsworthy, and D. W. Marsh. SPARK: A safety-related Ada subset. In W. J. Taylor, editor, *Ada in Transition, Proceedings of the Ada UK Conference.* IOS Press, October 1992.

[CH88] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CHN+94a] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. A review of existing refinement tools. Technical Report 94-8, Software Verification Research Centre, Department of Computer Science, University of Queensland, June 1994.

[CHN+94b] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. Refinement in Ergo. Technical Report 94-44, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, November 1994.

[CHN+95] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. Structured presentation of refinements and proofs. Technical report 95-46, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072, Australia, November 1995.

[CHN+96] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, and Jim Welsh. A tool for developing correct programs by refinement. In He Jifeng, editor, *Seventh Refinement Workshop, Workshop in Computing, BCS FACS*. Springer-Verlag, 1996. Also available as TR-95-49, Software Verification Research Centre, The University of Queensland.

[CJM+92] B. A. Carré, T. J. Jennings, F. J. Maclennan, P. F. Farrow, and J. R. Garnsworthy. SPARK — the SPADE Ada kernel. version 3.1. Program Validation Ltd, May 1992.

[COCD86] B. A. Carré, I. M. O'Neill, D. L. Clutterbuck, and C. W. Debney. SPADE: the Southampton Program Analysis and Development Environment. In I. Sommerville, editor, *Software Programming Environments*, pages 129–134. Peter Pereginus, 1986.

[CR90] D. A. Carrington and K. A. Robinson. Computer assistance for program refinement. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 312–321. Springer-Verlag, June 1990. 2nd International Conference, CAV'90, New Brunswick, NJ, USA.

[CR91] D. A. Carrington and K. A. Robinson. Refinement of two graph problems. In *Proceedings of the 4th Refinement Workshop*, pages 241–257, Cambridge, UK, 1991.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods, Formal Methods Europe*, number 873 in Lecture Notes in Computer Science. Springer-Verlag, April 1993.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and the formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs. 1976.

[dS89] J. L. A. Van de Snepscheut, editor. *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[FNG92] F. K. Hanna, N. Daeche, and G. Howells. Implementation of the Veritas Design Logic. In V. Stavridon, T. F. Melham, and R. T. Boute, editors, *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 77–94, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.

[GNU92] Lindsay Groves, Raymond Nickson, and Mark Utting. A tactic driven refinement tool. In Jones et al. [JSD92], pages 272–297.

[Gra91] Andrew Gravell. Specialising abstract programs. In Morris and Shaw [MS91], pages 34–50.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[Gru92] J. Grundy. A window inference tool for refinement. In Jones et al. [JSD92], pages 230–254.

[GW96] Marie-Claude Gaudel and James Woodcock, editors. *FME'96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe*, number 1051 in Lecture Notes in Computer Science. Springer-Verlag, March 1996.

[HD86] F. K. Hanna and N. Daeche. Purely functional implementation of a logic. In Jörg H. Siekmann, editor, *Proceedings of the 8th International Conference on Automated Deduction*, volume 230 of *LNCS*, pages 598–607, Oxford, UK, July 1986. Springer.

[HDL90a] F. K. Hanna, N. Daeche, and M. Longley. Veritas$^+$: A specification language based on type theory. In M. Leeser and G. Brown, editors, *Proceedings of the Mathematical Sciences Institute Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 358–379, Berlin, July 1990. Springer.

[HDL90b] F. Keith Hanna, Neil Daeche, and Mark Longley. Specification and verification using dependent types. *IEEE Transactions on Software Engineering*, 16(9):949–964, September 1990.

[HDNS96] Jonathon Hoare, Jeremy Dick, Dave Neilson, and Ib Søresen. Applying B technologies to CICS. In Gaudel and Woodcock [GW96], pages 74–84.

[Heh89] E. C. R. Hehner. Termination is timing. In de Snepscheut [dS89], pages 36–47.

[Heh90] E. C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14(2-3):133–158, 1990.

[Hes94] W. H. Hesselink. Nondeterminism and recursion via stacks and games. *Theoretical Computer Science*, 124:273–295, 1994.

[HH97] C. Michael Holloway and Kelly J. Hayhurst, editors. *Fourth NASA Langley Formal Methods Workshop*. September 1997. NASA Conference Publication 3356.

[HHJ+87] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[Hoa84] C. A. R. Hoare. Programs are predicates. *Philosophical Transactions of the Royal Society of London – Series A: Mathematical and Physical Sciences*, 312(1522):475–462, October 1984.

[Hoa86] C. A. R. Hoare. Maths adds safety to computer programs. *New Scientist*, pages 53–56, 18 September 1986.

[Hoa96] C. A. R. Hoare. How did software get so reliable without proof? In Gaudel and Woodcock [GW96], pages 1–17.

[Hor79] J. J. Horning. Verification of Euclid programs. In F. L. Bauer and M. Broy, editors, *Program Construction: International Summer School*, number 69 in Lecture Notes in Computer Science. Springer-Verlag, 1979.

[HST97] Daniel Hazel, Paul Strooper, and Owen Traynor. Possum: an animator for the SUM specification language. Technical Report 97-10, Software Verification Research Centre, School of Information Technology, University of Queensland, February 1997.

[Jac94] D. Jackson. Abstract model checking of infinite specifications. *Lecture Notes in Computer Science*, 873:519–531, 1994.

[JM93] Bart Jacob and Tom Melham. Translating dependent type theory into higher order logic. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications: Proceedings of the International Conference*, Lecture Notes in Computer Science, pages 209–229, Utrecht, March 1993. Springer-Verlag.

[Jon86] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, 1986.

[JSD92] Cliff B. Jones, Roger C. Shaw, and Tim Denvir, editors. *Proceedings of the 5th Refinement Workshop*, Workshops in Computing. BCS FACS, Springer-Verlag, January 1992.

[KM95] Steve King and Carroll Morgan. Exits in the refinement calculus. *Formal Aspects of Computing*, 7(1):54–76, 1995.

[Knu84] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[LGH+78] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. *Acta Informatica*, 10:1–26, 1978.

[LP98] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? SRC Research Report 147, Digital Systems Research Center, April 1998.

[LRvW95] Thomas Långbacka, Rimvydas Rukšėnas, and Joakim von Wright. TkWinHOL: A tool for doing window inference in HOL. In E. Thomas Schubert, Philip J. Windley, and James Alves-Foss, editors, *Higher Order Theorem Proving and its Applications: Proceedings of the 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 245–260, Aspen Grove, Utah, September 1995. Springer-Verlag.

[LS79] David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, October 1979.

[Mar95] D. W. R. Marsh. A mechanised language for program refinement. Master's thesis, University of Southampton, April 1995. Minithesis submitted for upgrading from MPhil to PhD.

[Mar97] Andrew Martin. Approaches to proof in Z – or – why effective proof tool support for Z is hard. Technical Report 97-34, Software Verification Research Centre, University of Queensland, November 1997.

[MG90] C. C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990. Reprinted in [MV94].

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[MO94] D. W. R. Marsh and I. O'Neill. Formal semantics of SPARK. Program Validation Ltd., March 1994.

[Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[Mor88a] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988. Reprinted in [MV94].

[Mor88b] Carroll C. Morgan. Data refinement using miracles. *Information Processing Letters*, 26(5):243–246, January 1988. Reprinted in [MV94].

[Mor88c] Carroll C. Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1):17–28, 1988. Reprinted in [MV94].

[Mor89] C. C. Morgan. Types and invariants in the refinement calculus. In de Snepscheut [dS89], pages 363–378. Reprinted in [MV94].

[Mor94] Carroll Morgan. *Programming from Specifications.* Prentice Hall International, second edition, 1994.

[MR87] Carroll Morgan and Ken Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5), September 1987. Reprinted in [MV94].

[MS91] Joseph M. Morris and Roger C. Shaw, editors. *Proceedings of the 4th Refinement Workshop*, Workshops in Computing. BCS FACS, Springer-Verlag, January 1991.

[MV94] Caroll Morgan and Trevor Vickers, editors. *On the Refinement Calculus.* Springer-Verlag, 1994.

[NH96] Ray Nickson and Ian Hayes. Supporting contexts in program refinement. Technical report 96-29, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072, Australia, December 1996.

[OAK97] C. O'Halloran, R. Arthan, and D. King. Using a formal specification contractually. *Formal Aspects of Computing*, 9(4):349–358, 1997.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, number 607 in Lecture Notes in Computer Science, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[ORSH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[OS97] Sam Owre and Natarajan Shankar. The formal sematics of PVS. Technical Report CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA, August 1997.

[OS98] C. O'Halloran and A. Smith. Don't verify, abstract! In *Thirteenth International Conference on Automated Software Engineering*, pages 53–62. IEEE Computer Society Press, 1998.

[OS99] C. O'Halloran and A. Smith. Verification of picture generated code. In *14th IEEE International Conference on Automated Software Engineering*, pages 127–136. IEEE Computer Society Press, 1999.

[OSR93] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park CA 94025, April 1993.

[OSRSC98] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, September 1998.

[OSS95] C. M. O'Halloran, C. T. Sennett, and A. Smith. A commentrary on the specification of the compliance notation for SPARK and Z. Technical Report DRA/CIS(SE2)/PROJ/SWI/TR/1/1.1, Defence Evaluation and Research Agency, St. Andrews Road, Malvern, Worcestershire, November 1995.

[OSS97] C. M. O'Halloran, C. T. Sennett, and A. Smith. Specification of the compliance notation for SPARK and Z. Technical Report DRA/CIS/CSE3/TR/94/27/3.0, Defence Evaluation and Research Agency, St. Andrews Road, Malvern, Worcestershire, September 1997.

[PE97] John Peterson and Kevin Hammond (Editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.4). Technical report, Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1105, February 1997.

[PP94] D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 58–65, August 1994.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, pages 97–105, September 1994.

[Rus93] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-07, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, November 1993.

[RvW97] Rimvydas Rukšėnas and Joakim von Wright. A tool for data refinement. Technical Report 119, Turku Centre for Computer Science, Åbo Akademi University, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, August 1997.

[Saa97] Mark Saaltink. Domain checking Z specifications. In *[HH97]*, 1997.

[SC96] P. Stock and D. Carringtion. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 11(22):777–793, November 1996.

[Sen92] C. T. Sennett. Demonstrating the compliance of Ada programs with Z specifications. In Jones et al. [JSD92], pages 70–87.

[Sha93] N. Shankar. Abstract datatypes in PVS. Technical Report CSL-93-3, Computer Science Laboratory, SRI International, Menlo Park CA 94025, December 1993.

[Smi96] A. Smith. Examples of compliance arguments between SPARK and Z. Technical Report DRA/CIS3/PROJ/SWI/TR/12, Defence Evaluation and Research Agency, St. Andrews Road, Malvern, Worcestershire, September 1996.

[Spi88] J M Spivey. *Understanding Z: a specification language and its formal semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 1989.

[SS99] E. Sekerinski and K. Sere, editors. *Program Development by Refinement: Case Studies Using the B Method.* Formal Approaches to Computing and Information Technology. Springer-Verlag, 1999.

[Tho92] Simon Thompson. Are subsets necessary in Martin-Löf type theory? In J. P. Myers Jr and M. J. O'Donnell, editors, *Constructivity in Computer Science*, volume 613 of *Lecture Notes in Computer Science*, pages 46–57. Springer-Verlag, January 1992.

[Ver92] Guide to the Veritas design logic (standard ML version). Final Draft, Revision 4. Available from www.ukc.ac.uk. Appears incomplete, June 1992.

[Vic90] T. Vickers. An overview of a refinement editor. In *Proceedings of the Fifth Australian Software Engineering Conference*, pages 39–44, 1990.

[Wad92] P. L. Wadler. The essence of functional programming. *Principles of Programming Languages, ACM*, January 1992.

[War92] Martin Ward. A recursion removal theorem. In Jones et al. [JSD92], pages 43–69.

[War96] Martin Ward. Derivation of data intensive algorithms by formal transformation: The Schnorr-Waite graph marking algorithm. *IEEE Transactions on Software Engineering*, 22(9):665–685, September 1996.

[Wir71] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, 1971.

[Woo91] Kenneth R. Wood. The elusive software refinery: a case study in program development. In Morris and Shaw [MS91], pages 281–325.

[Wri92] J. von Wright. The lattice of data refinement. Reports on Computer Science & Mathematics Ser. A. 130, Åbo Akademi, February 1992.

[Wri94] J. von Wright. Program refinement by theorem prover. Reports on Computer Science & Mathematics Ser. A. 146, Åbo Akademi, January 1994. Also in the 6th Refinement Workshop.

[WSH81] J. Welsh, W. J. Sneeringer, and C. A. R. Hoare. Ambiguities and insecurities in Pascal. In D. W. Barron, editor, *Pascal — The Language and its Implementation*, chapter 2, pages 5–19. John Wiley & Sons, Ltd., 1981.