COMPUTER SYSTEMS LABORATORY

DEPARTMENTS OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
STANFORD UNIVERSITY. STANFORD, CA 94305-2192

A Strongly Typed Language for Specifying Programs

Friedrich W. von Henke

Technical Report No. 84-258

Program Analysis and Verification Group Report No. 23

January 1984

‘his work was supported by the Advanced Research Projects Agency, Department of Defense, under contracts MDA-303-80-C-0159 and NOO-039-82-C-0250.

A Strongly Typed Language for Specifying Programs

Friedrich W. von Henke

Program Analysis & Verification Report No. 23

Technical Report No. 84-258

January 1984

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

A language for specifying and annotating programs is presented. The language is intended to be used in connection with a strongly typed programming language. It provides a framework for the definition of specification concepts and the specification of programs by means of assertions and annotations. The 'language includes facilities for defining concepts axiomatically and to group definitions of related concepts and derived properties (lemmas) in theories. All entities in the language are required to be strongly typed; however, the language provides a very flexible type system which includes polymorphic (or generic) types. The paper presents a type checking algorithm for the language and discusses the relationship between specification language and programming language.

Key Words and Phrases: Program Specification, Specification Language, Generic Data Types, Type Checking.

Table of Contents

1. INTRODUCTION

This paper proposes a language to bc used in the specification and analysis of programs. The specification language is intended for use in conjunction with a strongly typed programming language. By "specification language" we mean a language that allows a programmer to define specification concepts independantly of programs and then use those concepts to specify intended propcrties of programs by, e.g., formal assertions.

In comparison with most programming languages, the specification language itself is fairly simple. It includes facilities for defining specification concepts axiomatically or by explicit definitions, and for grouping related concept definitions into logically and semantically meaningful units. The specification language is strongly typed. However, the type system of the language is more complex than that usually provided in programming languages; major emphasis is givcn in this paper to the motivation and development of an appropriate type system for the specification language.

Many modem high-level programming languages like Pascal and Ada[1] [1] require that each use of an identifier be preceded by a declaration indicating its type, thus allowing detection at compiletime of some inconsistent and erroneous uses. In contrast, verification systems like the Stanford Pascal Verifier [5] do not impose the same strict discipline of strong typing on the specification language. Experience has shown that type checking could be just as beneficial for catching errors in assertions as in programs; for instance, even simple spelling errors often remain unnoticed until a tedious analysis of an incompleted proof reveals them. . On the other hand, strong typing can at times get in the way of natural and concise expression by being overly restrictive.

The type system for the specification language proposed hcre attempts to strike a balance between the benefits of strong typing and ease of expression. It requires that all identifiers used in the specification language have a type assigned to them by declaration, with the exception of local variables. The type system includes *generic* or *polymorphic* types in order to avoid having to declare instances of type structures and operators that are common to whole families of types. For example, generic types make it possible to talk about arrays in general without having to specify the array size or the type of array elements. The type of an identifier need not be unique, i.e., we allow a restricted form of overloading of identifiers.

1.1 THE ROLE OF TY PES IN FORMAL SPECIFICATION

Asscrtions refer to program variablcs. Thus, types in assertions have. to be bascd on thc types assigned to variables by the programming language. However, thc type system of thc programming language is not quite appropriate for assertions (and should not be adopted for assertions without modifications): the role of types in specification languages differs somewhat from their role in programming languages.

1. In specification languages, a type is primarily associated with a *theory* in which the reasoning

[1]Ada is a registered trademark of the U. S. Government, Ada Joint Program Office

about objects of that type is done. Accordingly, types must be disjoint, in order to uniquely identify the theory an object (or value) belongs to. For instance, the theory of integers is not a subtheory of the theory of rational numbers (nor vice versa). Thus, the types *integer* and *real* are kept completely separate in the context of formal reasoning. On the other hand, there is only *one* theory for reasoning about integers: programming languages like Ada may distinguish different integer types that are equivalent ("derived types" in Ada), but in the specification language these types are associated with the same specification type and the same theory. (In addition to a domain of values, programming language types express *ownership* of values, which is not relevant for reasoning about values.)

2. The type of an object must be determinable by purely syntactic means. This requirement contrasts with constraints on programming language types which typically require run-time checks, or else some form of theorem proving is required to determine at compile-time whether they are satisfied. For instance, a range or subset (e.g., of integers) cannot constitute a separate type; the information contained in constraints must be expressed explicitly by adding constraint predicates to assertions. In short, a specification language type stands for a whole family of programming language types.

3. There is no notion of *block structure* for theories. Rather, declaration of symbols is cummulative. Thus it is not possible to "redeclare" a type or function in an inner scope.

4. A type in the specification language can be abstract in the sense that it does not define a domain of'values associated with the type. Thus a type can be declared simply by introducing its name.

5. A type and the associated theory may be generic; functions in such a theory are normally declared with generic parameter and result types. In contrast to many programming languages, entities with generic types need not always be fully instantiated; in particular, generic functions can be used without first creating an instance for particular, fully instantiated types.

The type system proposed here reflects the particular needs of a typed specification language that is intended to complement a strongly typed programming language. The relationship between specification language types and programming language types is discussed in Section 5.

1.2 OVERVIEW

The main unit in the proposed specification language is the *theory*. A theory packages related object types, operations and specification concepts in form of declarations and descriptions of concept properties. More specifically, a theory contains declarations for related types, constants and function symbols, (explicit) definitions of functions and implicit descriptions of concepts in form of axioms and lemmas. Theories provide a framework for organizing such entities by grouping them together; the theory name then permits to refer to them collectively. In contrast to programming languages, the aspects of encapsulation and information hiding usually associated with the concept of program module (or package) are not relevant here.

As a simple example, consider a (generic) theory of arrays with unspecified index and element type:

```
theory Arrays (α1,α2):                  -- α1 and α2 are type parameters to the theory
     type array (α1,α2);                -- in morecommon notation: array(α1) of α2
     function astore (array(α1,α2),α1,α2): array(α1,α2);
     function aselect (array(α1,α2),α1): α2;
     var A   : array(α1,α2);            -- These variable declarations are optional:
                                        -- they apply only to the axioms.
     var i , j : α1;
     var e   : α2;
     axiom Al: aselect(astore(A,i,e),j) = if i=j then e else aselect(A,j);
     axiom AZ: astore(A,i,aselect(A,i)) = A;
end Arrays:
```

The theory has two type parameters, α_1 and α_2, which stand for the types of indices and array elements, respectively. It introduces a parametrized type array and generic functions for array update and selection whose properties are described by the axioms. The variable declarations are redundant as the types of the variables in the axioms can be deduced from the types of the functions, as discussed in Section 4.

In this paper, major emphasis is given to the development of an appropriate type system and the mechanics of type checking. The remainder of the paper is organized in 4 sections. Section 2 introduces syntax and semantics of types in the specification language. Section 3 deals with the concept of "theory" and declaration and decription of concepts. Section 4 discusses type checking in the presence of generic types and overloading. Some relationships between the specification language and the programming language Ada, in particular the relationship between the respective type systems, are discussed in Section 5; this section also gives an extended example of a program with specifications.' Appendix A contains a BNF description of the syntax used in the main part of the paper; some particulars, like the definition of infix operators, are left unspecified. Appendix B gives a specification theory for pointer types (or access types in Ada) and pointer struc tures.

2. TYPES

2.1 TYPE EXPRESSIONS

Type expressions are thc syntactic entities that denote types. In most contcxts it is not necessary to maintain a clear distinction between "types" and "type expressions", and we will use the term "type expression'* in the rest of the paper only when syntactic aspects are to be emphasized (as, for instance, in this section).

The following are type expressions:

1. anidentifier: integer;
2. an *enumeration* (list of identifiers): (a, b, c). The elements of the list must be pairwise distinct.
3. a *type variable:* type variables are denoted by $a, \alpha_1, \alpha_2, \ldots$.
4. a composite exprcssion of the form $c(\tau_1, \tau_2, \ldots, \tau_n)$, where $\tau_1, \tau_2, \ldots, \tau_n$ are type expressions and c is the name of a previously declared *generic type* (type constructor, see below) with n type parameters;
5. a *record* type: $\texttt{record}(s_1:\tau_1; \ldots; s_n:\tau_n)$, where $\tau_1, \ldots, \tau_n$ are type expressions and $s_1, \ldots, s_n$ are identifiers.

Type expressions and types may be constant or generic. A *constant type expression* does not contain any occurrence of a type variable; a type expression that contains type variables *is generic.* A *constant type* is a type denoted by a constant type expression, 'and a *generic type* is denoted by a generic type expression. A type expression that is not just a type variable - and similarly the type denoted by it - is called *proper.* In the remainder of the paper, type expressions may always be generic unless stated otherwise.

A generic type expression τ_1 can be *specialized* to another (possibly generic) type expression τ_2 by substituting type expressions for one or more type variables in τ_1; here substitution has the usual meaning that *all* occurrences of a type variable are rcplaced by the *same* type expression. The result of specializing a type expression τ_1 is an *instance* of τ_1. A type expression τ_2 is a *constant instance* of τ_1 if τ_2 is constant and an instance of τ_1. The term *instantiation* is reserved for a specialization to a constant type expression. In general, all type expressions in a substitution are proper (in this case, the specialization and the resulting instance are also called proper, for emphasis), but mere renaming of type variables is includcd for the sake of completeness. In the usual manner of not distinguishing between types and type expressions. we will also talk of a *type* as being an instance of another type etc.

A constant type expression that does not consist of just an identifier is similar to a structured type in Pascal or Ada, with the main difference that those languages provide only a fixed number of type construction schemes whereas the specification language allows users to define their own type constructors.

Examples of Type Expressions.

```
integer
(a,b,c,d,e,f)                  -- an enumeration
array(α1,α2)                   -- a generic type
array(integer,array(integer,α))
```

2.2 TYPEDECLARATIONS

An identifier denotes a type if it has been declared in a type declaration. A *type declaration* has one of the following forms:

```
(1) type <type-name> ;
(2) type <type-name> = <type-expression> ;
(3) type <type-name> (<type_variable_list>) ;
(4) type <type-name> (<type-variable-list>) = <type-expression> ;
```

where `<type_name>` is the identifier being declared, and the type expression in forms (2) and (4) may be generic.

A declaration of form (3) or (4) introduces a type name that denotes a generic type. As in ordinary function declarations, the type variables in the parameter list must be pairwise distinct. In form (3), the list of type variables merely indicates the number of formal type parameters. In a declaration of form (4), the type expression may mention only type variables from the type variable list on the left-hand side of the defining equation.

Type declarations may not be recursive, with the exception of recursion that uses explicit pointer types (as permitted in languages like Ada or Pascal); only the declaration of a pointer type may refer to a type that has not been declared yet (cf. Appendix B).

A type declaration of form (1) introduces a new constant type that is different from any other type. Similarly, a declaration of form (3) declares the name of a new generic type. We will sometimes use the term *type constructor* for generic type names in order to emphasize that it is a way of constructing new types (e.g., array types) from parameter types.

A declaration of form (2) or (4) introduces a new name for the type denoted by the type expression, which in general is a specialization of a generic type. It does *not* declare a new, distinct type; rather the new name and the type expression arc synonyms for thc same type.

A declaration of form (4) is used to introduce a specialized generic type. For example, a declaration

```
type iarray(α) = array(integer,α);
```

declares a generic type of arrays for which the type of indices is fixed to the type `integer`.

The difference between declarations of form (2) and form (4) is that for a declaration of form (2) it is hidden whether thc declarcd typc is gcncric; thus it cannot bc specialized, cvcn though the dcfining typc cxprcssion may contain type variables. For example, the dcclarations

```
type LL_1 = list(list(α));
type LL_2(α) = list(list(α));
```

introduce the same type, but the syntax allows only $LL_2(\alpha)$ to be specialized further; for instance, by the declaration

```
type LL_3 = LL_2(integer);
```

LL_3 becomes a synonym for the type expression `list(list(integer))`.

All type variables denote an arbitrary type ("any") and thus are semantically equivalent. Occurrences of the same type variable in one type expression denote the same (but unspecified) type; for example, in an instance of `comp1(α, α, α)` all occurrences of α have to be specialized to the same type, whereas `comp2(α_1, α_2)` could also be specialized to, e.g., `comp2(integer, boolean)`. The only reason for having more than one variable is this ability of expressing which argument types may differ or have to be the same.

Note. Another way of looking at type variables is to regard them as parameters that are "bound" by a special lambda-operator for types. This point of view also explains that all type variables are semantically equivalent; they are just place holders and can be renamed arbitrarily (similar to a-conversion in lambda-calculus).

A type declared with an enumeration as type expression is called *enumeration type.* Enumeration types have a meaning similar to that in Pascal or Ada. The elements of an enumeration type (which must be pairwise distinct) are implicitly declared as constants.

Type equality. Two type expressions are equivalent if both can be reduced to the same type expression, taking type definitions (synonyms) into account and permitting renaming of type variables (cf. Section 4.2). Equivalent type expressions denote the same type. Thus, in contrast to many programming languages (like Ada), the specification language uses *structural type equality.* For the kind of parametrized type definition used here, structural equivalence of type expressions is decidable (see, e.g., [6]).

Examples of Type Declarations.

```
type integer;
type A;                             -- The type A is distinct from any other type:
                                    -- nothing is known about its domain of values or structure
                                    -- without further specifications.
type C = integer;                   -- C is a synonym for integer
type lets = (a,b,c,d,e,f);          -- lets denotes an enumeration type
type array(α_1,α_2);                -- a generic array type: α_1: type of indices,
                                    -- α_2: type of elements
type arr1(α) = array(α,α);          -- a generic array type for which index and element
                                    -- type are the same
type intarr1 = arr1(integer);       -- the constant type of arrays with integer indices and
                                    -- integer elements
type intarr2 = arr2(C);             -- intarr2 denotes the same type as intarr1
type two_arr(α_1,α_2,α_3) = array(α_1,array(α_2,α_3));
                                    -- a two-dimensional generic array type
```

2.3 REMARKS ONTHETYPESYSTEM

Abstract Types. A type declared in a declaration without a defining type expression (form (1) or (3)) is opaque, i.e., nothing is known about its structure and its elements. Such types are used to model user-defined abstract types. Information about abstract types is normally expressed in axioms (see Section 3.2).

Recursive Types. The proposed system of types has been tailored to the purpose stated in the introduction. It is not the most general model that appears to be useful in any context; in particular, abstract recursive types have not been included. In order to accomodate all types present in Pascal and Ada the system permits the limited form of recursion in the declaration of "concrete" recursive data structures provided in those languages. We assume that abstract recursive types are declared as opaque types (forms (1) and (3)) and that their structure is expressed indirectly, namely by axioms; see Section 3 for examples. In a context where data structures are actually used in proofs (e.g., for defining structural induction) a form of recursive type definition that directly indicates the data structure seems desirable; the present framework can then be extended to include those.

Record Types. Record types have essentially the same meaning as in Pascal or Ada. It would seem desirable to permit generic record types. This, however, cannot be achieved in full generality without extending the notion of type expression.

A record structure can be regarded as a Cartesian product of record fields. Accordingly, a generic record is a . generalized Cartesian product of indeterminate dimension. It is therefore not possible to define a type constructor record as one taking type expressions as arguments, similar to, say, the type constructor ar ray; rather, the proper fimctionality requires introduction of "families of types".

A family of types is a mapping from an enumeration type into the domain of types. Let E be an enumeration type and let $<\tau_e | e \varepsilon E>$ denote a family of types over E. Then we can express the constructor record as

$$\text{record}(E, <\tau_e | e \varepsilon E>)$$

Alternatively, record could be used as type constructor with fixed field selectors; for instance, the declaration

type t = $\text{record}(a{:}\alpha_1;\ b{:}\alpha_2;\ c{:}\alpha_3);$

defines a (generic) record type with three fields. This method does not cause the complication mentioned above, but also appears to be less useful.

Apart from concise statement of properties of record update and selection, the usefulness of generic rcdords is questionable. Record structures are primarily an implementation structure; in specifications it appears more desirable to express concepts in terms of abstract types and functions operating on them. In light of this the proposed type system does not include any particular definition method for generic records.

3. THEORIES

Theories are the entities that contain descriptions of user-defined concepts. A theory introduces (in declarations) the names of related types, constants and functions that are used to express concepts, and states their properties in the form of definitions, axioms and lemmas. *Definitions* are explicit, possibly recursive, function definitions. *Axioms* and *lemmas* are arbitrary quantifier-free first-order formulas.

A theory may have types as parameters; a parametrized theory is called *generic.* The type parameters of a theory are indicated by a list of type variables in the header of the theory. Only those type variables that appear in the parameter list may be used in the theory body. The typc parameters are the only type variables that may be used in the theory body.

A theory often defines what is called an *abstract data type,* introducing one new data type and a set of operations on it (cf. the examples below). However, the notion of theory used here is more general than that of (parametrized) abstract data types; it rather cornprizes any semantically meaningful collection of related types and concepts, thus corresponding more closely to the notion of a theory in first-order logic.

A theory is meant to be a self-contained unit. This requires that it be a closed scope: in general, no entity (type, function, or constant) can bc used unless declared within that unit. The mechanism of theory extension (see Section 3.4) provides the means for importing entities declared in another theory, thus' making them visible in the importing theory. For the sake of convenience the elementary theory Booleans given below is considered universally visible (or implicitly imported) in every theory.

A theory is identified by its name. The form of a theory declaration is as follows.

```
theory <theory-name> [(<type-parameters>)] :
     [<extends-clause>]
     <theory-body>
end <theory-name>;
```

The meaning of <extends-clause> is described below in Section 3.4. The <theory-body> may , contain declarations, definitions, axioms and lemmas, as described in the following sections.

3.1 DECLARATIONS

In general, names have to be declared before they can be used. This rule does not apply to variables, which are trcatcd specially, and prc-dcclarcd cntitics. Type declarations havc bccn discussed in Section 2. This section deals with declaration of functions, constants and variables. Unless stated otherwise, all types may be generic, i.e., contain type variables.

Function, constant and variable declarations are preceded by the respective keywords, function, constant or **var.**

Functions are declared by listing the type of paramctcrs and result: thc declaration function f $(\tau_1,\ldots,\tau_n)$: τ_{n+1};

means that f is an n-ary function which takes arguments of types $\tau_1,\ldots,\tau_n$ and returns a value of type τ_{n+1}.

Functions may also be declared as part of a definition; see Section 3.3 below. For each type, an equality relation is assumed to be available (i.e. predcfined) and thus needs not be introduced explicitly.

Constants are nullary functions; their declaration contains only one type, the "result" type. The declaration of a constant entails that it is distinct from any previously declared constant.

Predicates are simply boolean-valued functions. Types, functions and constants are the *symbols* of a theory.

A *variable* is declared as in Pascal; for instance:

var x: τ;

where τ is a type expression. Declaration of *variables* in theories is optional. As will be explained below, variable declarations are not required for type checking, but they may be added for improved documentation.

Generic functions and constants. A function or constant is called *generic* if its declared type is generic. In a generic function declaration, the type variables occurring in the result type normally also occur in one of the argument types so that for every instance of the function the type of the result is completely determined by the types of the arguments. In the case of generic constants (viewed as nullary functions) this is not possible; however, as generic constants are obviously useful (cf. null_list, nil below) we do not in general require that the result type contain only type variables that occur in one of the argument types.

For the scope of declarations see Section 3.5 below.

Examples.

```
function plus (integer,integer): integer:      -- declares a binary function on integers
function less-eq (integer,integer): boolean;     -- a binary predicate on integers
function length (llist(α)): integer:
                              -- a generic function mapping arbitrary linear lists into integers
function append (llist(α),llist(α)): llist(α); -- abinarygenericfinction

constant zero: integer:
constant empty: set(a);              -- a generic constant
constant null-list: llist(α).;       -- another generic constant
```

The following is the declaration part of an elementary generic theory of linear lists.

```
theory Linear-lists (a):
     type llist (a);              -- the generic list type introduced by the theory

     function add (α,llist(α)): llist(α);
     function head (llist(α)): a;
     function tail (llist(α)): llist(α);
     constant null-list: llist(α);
end Linear-lists;
```

3.2 AXIOMS AND LEMMAS

An *axiom* has the form

```
axiom <name> : <expression>;
```

where <expression> is an arbitrary well-formed quantifier-free first order formula. Syntactically, <expression> is an expression of type boolean which is built up from previously declared functions and constants, including boolean connectives, equality, conditionals and variables (see Appendix A). The expression is interpreted as universally quantified over all variables occurring free in it.

A *lemma* has the same form as an axiom, except for the leading keyword lemma.

The distinction between axioms and lemmas is as one should expect: axioms define concepts axiomatically, and lemmas state derived properties. The distinction is also relevant for showing that a theory is consistent. In a proper environment, lemmas should be formally proved before they are used; then they need not be taken into consideration when analyzing consistency of the theory. (A discussion of methods for demonstrating consistency of theories, e.g. via model construction, goes beyond the scope of this paper.)

Examples.

1. The theory Booleans assumed to be visible everywhere could be described as follows.

```
theory Booleans:
    type boolean;
    constant true: boolean;
    constant false: boolean:
    function and (boolean,boolean): boolean;
    function or (boolean,boolean): boolean;
    function implies (boolean,boolean): boolean:
    function not (boolean): boolean;

    axiom a: and(true,true)=true ∧ and(true,false)=false ∧ . . . .
    . . .      -- The axioms give the truth tables for the boolean operations. Note that the
               -- inequality true≠false need not be added to the axioms as it is implied by the
               -- constant declarations.
end Booleans;
```

In what follows we will use infix operators ∧, ∨ and ⊃ and the symbol ¬ to denote the functions declared above.

2. The following theory gives an (incomplete) axiomatization of integers.

```
theory Integers:
    type integer:
    constant 0: integer:
    constant 1: integer:
    function plus (integer,integer): integer:
    function minus (integer,integer): integer;

    axiom zero :  plus(x,0) = x;
```

```
        axiom one  :  plus(x,1) ≠ x;
        axiom corn :  plus(x,y) = plus(y,x);
        axiom inv  :  minus(plus(x,y),y) = x;
        ...
    end Integers:
```

3. The following axioms complete the theory Linear_lists.

```
    theory Linear-lists(α):
        type llist(α);
        function . . .                    -- see Section 3.1

        axiom head :  head(add(x, y)) = x;
        axiom tail :  tail(add(x, y)) = y;
        axiom add  :  add(x,y) ≠ null-list;
    end Linear-lists;
```

4. A simple theory for a structure of *binary trees* over an arbitrary domain of atoms is defined as follows.

```
    theory Binary-trees (α):
        type btree (α);

        function comp (btree(α),btree(α)): btree(α);
        function is-atom (btree(α)): boolean;
        function car (btree(α)): btree(α);
        function cdr (btree(α)): btree(α);
        function mktree (α): btree(α):
        function mkatom (btree(α)): α;

        var x,y: btree(α);               -- These variable declarations are included for
        var u  : α;                      -- demonstration purposes only.

        axiom car  :  car(comp(x,y)) = x;
        axiom cdr  :  cdr(comp(x,y)) = y;
        axiom comp :  ¬ is_atom(comp(x,y));
        axiom atom :  is_atom(mktree(u));
        axiom mkt  :  mkatom(mktree(u)) = u;
        axiom mka  :  is-atom(x) ⊃ mktree(mkatom(x)) = x;
    end Binary-trees;
```

Since an atom is ndt automatically a binary tree (the types α and btree(α) are disjoint) the conversion functions mktree and mkatom are necessary; the axioms mkt and mka indicate that the domain of atoms is embedded one-to-one into the domain of binary trees.

3.3 DEFINITIONS

A definition combines a declaration of a function with a defining equation. The general form of a definition is:

definition [<name>:] $f(x_1:\tau_1,..,x_n:\tau_n)$: τ_{n+1} := <expression>;

where naming of the definition is optional, and <expression> is an expression (see Appendix A) which may contain only $x_1, \ldots, x_n$ as free variables; the name f may occur in <expression>, i.e., the definition may be recursive. The declaration part (i.e. the part preceding ":=") is equivalent to an ordinary function declaration,

function $f(\tau_1, \ldots, \tau_n): \tau_{n+1}$;

If several parameters share the same type, they can be combined in the usual fashion by writing, e.g., $g(x,y:\tau_1):\tau_2$ instead of $g(x:\tau_1,y:\tau_1):\tau_2$.

The function parameters x_i are variables. For the purpose of type checking, x_i is regarded as declared with type τ_i; the scope of these declarations is <expression>.

A definition asserts an equation as an axiom; for instance, the definition given above asserts,

$f(x_1, \ldots, x_n)$ = <expression>

we call this the *defining equation* for the function f.

Note. As the scope of the declaration of function parameters is the definition, the same parameter name (variable) can be associated with different types in different definitions. However, it is good practice to use variable names consistently so that at least within one theory a name is always associated with the same type. associate a variable name consistently with the same type. This will at least make it easier for human readers to understand definitions.

In order to accommodate mutually redursive function definitions, a definition can have more than one declaration plus defining equation.

Examples.

1. The function append may be defined by

```
definition append (x,y:llist(α)): llist(α) :=
         if x=null_list then y else add(head(x),append(tail(x),y));
```

2. The following is an example of a named definition defining more than one function:

```
definition rev:
    reverse (x:llist(α)): llist(α) := rev_aux(x,null_list),
    rev-aux (x,y:llist(α)).: llist(α) :=
         if x=null_list then y else rev_aux(tail(x),add(head(x),y));
```

3.4 THEORY EXTENSIONS

A theory may be an extension of other theories. A theory can be extended by adding new declarations, definitions, axioms and lemmas. A theory is made an extension of other, previously defined, theories by including an **extends** clause,

extends <theory-name>, <theory-name>, ...

at the beginning of the theory definition. Inclusion of an extends clause is equivalent to including the text of the body of the named theory, with the exception that variable declarations occurring in the named theory are no longer effective. Accordingly, extensions have to be conservative; in particular, redeclaration of types and symbols is not permitted, except for certain kinds of overloading of function symbols (see Section 3.5). Furthermore, care must bc taken to avoid introducing inconsistencies by adding new axioms.

A generic theory can be extended in a specialized context without being specialized first (see Example 2 below); a generic theory is never explicitly specialized.

Examples.

1. We extend the theory of linear lists by adding some function definitions. The extended theory is generic in the same way the theory Linear_lists is.

```
theory Extended-linear-lists (α) :
    extends Linear-lists, Integers;

    definition append (x,y:llist(α)): llist(α) :=
        if x=null_list then y else add(head(x),append(tail(x),y));

    definition length (x:llist): integer :=
        if x-null-list then 0 else 1+length(tail(x)));

    definition rev:
        reverse (x:llist(α)): llist(α) := rev_aux(x,null_list),
        rev-aux (x,y:llist(α)): llist(α) :=
            if x=null_list then y else rev_aux(tail(x),add(head(x),y));
end Extended-linear-lists;
```

2. The following theory extends the theory Linear-lists for a specialized context in which the parameter type is instantiated to integer. Note that the symbols of the theory Linear_lists, i.e., head, tail and null_list are not explicitly specialized; rather their use is controlled by the requirements for type correctness of expressions (see Section 4).

```
theory Integer-lists:
    extends Linear-lists, Integers:

    type Int_list = llist(integer);

    definition sum(x:Int_List): integer :=
        if x=null_list then 0 else head(x)+sum(tail(x));
end Integer-lists;
```

3. An extension of the theory Integers by ordering predicates:

```
theory Ordered-Integers:
    extends Integers ;
    function less-eq (integer, integer) : boolean;
    definition less (x,y:integer): boolean := less_eq(x,y) ∧ x≠y;

    axiom 11: less-eq(x,x);
    axiom 12: less_eq(x,y) ∧ less-eq(y,z) ⊃ less-eq(x,z);
    axiom 13 : less_eq(0,x);
```

```
        axiom 14: less_eq(x,plus(x,1));
        axiom 15 : less_eq(x,y) ∨ less_eq(y,x);
    end Ordered-Integers;
```

In the remainder of the paper we use infix symbols ≤ and < for the functions less_eq and less.

4. The theory Permutation is a standard theory describing permutations of arrays.

```
    theory Permutation (α1,α2):
        extends Arrays ;    --Arrays is the theory given in Section 1.2.
                    -- The theory Permutation has two type parameters, which are
                    -- the same as those of the array theory, i.e. index type and element
                    -- type. We cannot express a requirement that α1 be a discrete type.
        type arr = array (α1,α2);        -- an abbreviating synonym

        function trans (A:arr; i,j:α1): arr;
        function perm (A,B:arr): boolean:

                    -- In the following we abbreviate select(A, k) to A(k) .

        axiom trans,ax:
            trans(A,i,j)(k) = if k=i then A(j) else
                                  if k=j then A(i) else A(k);

        axiom perm_ax:  -- perm is an equivalence relation generated by transpositions
            perm(trans(A,i,j), A) ∧
            ((perm(A,B) ∧ perm(B,C)) ⊃ perm(A,C));

        lemma perm_lm:  -- The following can be deduced from the axioms:
            perm(A, A) ∧
            (perm(A, B) ⊃ perm(B, A));

        -- Type checking of the axioms and lemmas will yield the variable! types
        -- A,B,C:arr and i,j:α1.
    end Permutation:
```

3.5 SCOPE OF DECLARATIONS, VISIBILITY AND OVERLOADING

The rules defining the scope of declarations and visibility in the specification language are set up so that consistent use of symbols is guaranteed.

The types, functions and constants declared in a specification theory - collectively called *symbols* for short - are the non-logical symbols of a theory of typed first-order logic in which proofs are carried out. Once a symbol has been introduced, its meaning cannot be changed; the nature of theories does not allow a block structure of scopes so that in different scopes a symbol may have different meanings. Visibility is therefore strictly cummulative; declared symbols cannot be hidden by other declarations as in block-structured languages. (As a consequence, it is not necessary to require a particular order of declarations within a theory, like a strict discipline of declaration before use, but such a discipline may be good style.) However, symbols can bc overloaded, i.e., the same name can be used to denote what amounts to several distinct logical symbols; the restrictions on overloading stated below guarantee that in any given context it is possible to identify which of the symbols that share a name is meant.

An extends-clause in a theory makes visiblc all symbols that are visible in the theories named in the clause. (As mentioned above, the theory `boo 1 e an s` is universally visible.) The rules for overloading apply equally to symbols madc visible by an extends-clause.

A symbol may be used with a type that is an instance of its (generic) declaration type; for details see Section 5.

Note. It might sometimes be useful to declare a new symbol with the specialized type; however, the language has no provisions for expressing that the new symbol is intended as a specialized instance of another symbol.

Variable Declarations

The type system is set up so that in general variables need not be declared in theories. If however, a variable declaration is included in a theory it is considered local to the theory, in contrast to declarations of symbols; in this case the theory acts like a "block" in a block-structured language. Every instance of a declared variable has to conform to the declaration.

Declaration of variables is not required except in particular situations of overloading (see below); its main purpose is documentation. Variables used in a theory, whether declared or not, are never visible outside of a theory. A variable declaration in a theory does not have any effect on any theory that includes - via an extends-clause - the theory containing the variable declaration; in this context, "theory" really means the syntactic unit.

Variables are treated differently from symbols so as to make it possible to use the same variable names in different contexts. Variables are not part of a theory. In theories, variables are merely placeholders for argument positions; they are irrelevant outside a theory (as syntactic unit).

Overloading

Under certain conditions, functions and constants may be redeclared. A new declaration overloads a previous onc; both declarations are visible. In this way the same symbol can be used in different context (this may be particularly useful for infix operators). However, in order to ensure that the type checking algorithm can identify a unique type for a given occurrence of an overloaded symbol, some restrictions must be imposed on overloading declarations.

The basic rule is that *the types of overloaded symbols must be disjoint.* Two types are disjoint if they cannot be specialized to the same type (or, in terms of the partial ordering on types discussed in Section 4.1, that no lub of the two types cxists). This means, in particular, that constant types are disjoint if they are distinct, and generic typcs built from distinct typc constructors are distinct. The types of two functions arc disjoint if the respcctive tuples of parameter types are disjoint, i.e if they differ in number of parameters or the respective parameter types are disjoint for at least one parameter position.

For example, an operator "*" could be overloaded as follows:

```
* (integer,integer): integer      -- integer multiplication
* (boolean,boolean): boolean      -- boolean "and"
* (α,llist(α)): llist(α)          -- add for linear lists
* (btree(α),btree(α)): btree(α)        -- comp for binary trees
```

However, overloading of the following functions,

```
* (α,llist(α)): llist(α)               -- add for linear lists
* (llist(α),llist(α)): llist(α)        -- append for linear lists
```

is not permitted as the types of the first parameter are not disjoint.

The basic rule is not quite sufficient in special situations where a a theory extends two disjoint generic theories. Consider the following situation: Assume the theories Linear_lists and Binary_trees have been extended to include the following functions:

```
in Extended-linear-lists:
    function * (llist(α),llist(α)): llist(α);   -- intended meaning: append
    function size(llist(α)): integer;           -- intended meaning: length of list

in Extended-binary-trees:
    function * (btree(a),btree(a)): btree(α);   -- intended meaning: comp
    function size (btree(α)): integer;          -- intended meaning: number of nodes
```

Now,let T be a theory which extends both Extended-linear-lists and Extended-binary-trees and includes a lemma,

```
theory T:
    extends Extended-linear-lists, Extended-binary-trees;
    ...
    lemma sl: size(a*b) = size(a)+size(b);
    ...
end T;
```

Lemma s 1 may be interpreted as being about linear lists or binary trees; there is no way to find out in T which is meant. However, if the lemma is interpreted over binary trees, it is incorrect; one would rather have

```
    lemma s2: size(a*b) = size(a)+size(b)+1;
```

Thus, in order to maintain consistency we must require that an expression like size(a*b) occur only in a context in which it is clear which type is meant. In the theory T we have to provide extra information in order to disambiguate statements like lemma s 1: it is necessary to include variable declarations for a and b in the theory T.

4. TYPE CORRECTNESS AND TYPE CHECKING

In this section we discuss what the notions “type correctness" and “type checking” mean in the presence of generic types. The approach taken here is strongly influenced by [4].

4.1 TYPE MATCHING, TYPE EQUIVALENCE, AND TY PE CORRECTNESS

Type Matching. The presence of generic types leads to a natural partial ordering on types. Let a relation $\leq$ hold between two type expressions τ_1 and τ_2, $\tau_1 \leq \tau_2$, if τ_2 is an instance of τ_1, i.e., if there exists a substitution π of type expressions for type variables in τ_1 such that $\tau_2 = \pi(\tau_1)$. Intuitively, the relationship $\tau_1 \leq \tau_2$ means that τ_2 denotes a *more specific type than* τ_1. We *say* that a type τ_2 *matches* another type τ_1 if $\tau_1 \leq \tau_2$. Any type matches a type variable, i.e. $\alpha \leq \tau$ holds for arbitrary type variable a and type expression τ.

For example, the following relationships hold:

```
a ≤ integer,
α1 ≤ α2
a ≤ llist(α) ≤ llist(llist(α)),
llist(α) ≤ llist(boolean),
llist(α) ≤ llist(α1);
```

but

```
integer doesnotrnatch boolean,
llist(α1) doesnotmatch array(α1,α2),
llist(boolean) doesnotmatch llist(α).
```

Type Equivalence. The relation $\leq$ is transitive and reflexive, thus a pre-ordering on type expressions. It is made a partial ordering by defining that two type expressions τ_1 and τ_2 are *equivalent,* $\tau_1 \equiv \tau_2$, if $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$, or if both τ_1 and τ_2 can be reduced to the same type expression by expanding declared synonyms (see Section 3.2). Equivalence of type expressions on the basis of the relation $\leq$ alone is possible only if the substitutions involved are mere renamings of type variables (“alpha-conversion” in lambda-calculus). In particular, all type variables arc equivalent as type expressions. Equivalent type expressions denote the same type, i.e they denote types that cannot be distinguished semantically; it is therefore justified to think of the the relation $\leq$ as a partial ordering on types.

As indicated by thc examplcs, the type variables are minimal with rcspcct to I, and constant types are maximal elements. A least upper bound (lub) of two types τ_1 and τ_2 with rcspect to $\leq$ is obtained by applying the “most general unifier” (which is unique up to rcnaming of type variables) if it exists; obviously, a lub of two types does not always exist.

The partial ordering $\leq$ is the basis for a precise definition of type correctness and type checking in thc presence of generic types.

Type Correctness. A *type assignment* tau assigns to each occurrence of each identifier in an expression a type which is an instance of the generic type of the identifier; such a type assignment also assigns specialized types to subexpressions. A type assignment tau for an expression e is *consistent* if it the expression e is type correct in the ordinary sense when all assigned types are treated as constants.

For example, let f be a generic function with type $(\tau_1, \ldots, \tau_n):\tau_{n+1}$, and for $i = 1, \ldots, n$ let σ_i be the type assigned to the expression e_i. Assigning the type σ_{n+1} to the expression $f(e_1, \ldots, e_n)$ will make the latter type consistent with respect to this type assignment provided the following conditions are met:

1. σ_i matches τ_i, i.e., $\tau_i \le \sigma_i$, for $i = 1, \ldots, n+1$.
2. the σ_i satisfy the equality constraints of the τ_i, i.e., occurrences of the same type variable in the τ_i are specialized to the same type in the σ_i;
3. σ_{n+1} is derived from τ_{n+1} by substituting for type variables the unique specialized instances determined by 1 and 2.

A expression e is *type correct* if there exists a consistent type assignment for e.

Note that every occurrence of a generic function or constant in an expression can require a different specialization of its (generic) type in order to make the expression type correct.

4.2 TYPE EVALUATION AND TYPE CHECKING

The process of type checking involves finding a type assignment that makes an expression type consistent. A type assignment can be constructed by building up "constraints" on the specializations of type variables.

We discuss type checking in three steps, each dealing with an increasingly complex situation.

1. *Type Evaluation.* When all variables occurring in an expression have a known and constant type, type checking amounts to straightforward evaluation. This process is similar to type checking for strongly typed programming languages with generic operators. Type evaluation is sufficient for type checking of assertions and definitions.
2. *Building up Type Constraints.* In the presence of undeclared variables, type checking involves inferring types for variables (finding a type assignment).
3. *Overloading.*

1. Type Evaluation. In the case that all variables occurring in an expression have a known and constant type, type checking amounts to a simple evaluation process. This process can be explained best inductively.

1. By assumption, variables have constant types.
2. Let $f(e_1, \ldots, e_n)$ be a expression where f is declared with type $(\tau_1, \ldots, \tau_n):\tau_{n+1}$, and assume that for $i = 1, \ldots, n$ the type assigned to the expression e_i is σ_i. If for each $i = 1, \ldots, n$ the type σ_i matches τ_i we get an instance σ_{n+1} of τ_{n+1} which is the type assigned to the whole expression. In the case where a subexpression e_i is a generic constant, we retroactively assign it a "fitting" specialized type in the manner described below.

3. If σ_i does not match τ_i for some i, or if an equality constraint is not met, then a typing error has been found and type evaluation stops.

Type Checking of Definitions. Definitions can be checked for type correctness by mere type evaluation. The types of parameters of the defined function are "locally constant", i.e., for the purpose of checking type correctness of the defining equation all parameter types are treated as constant types, and the parameter variables have a defined type. The defining equation in a definition

definition <name>: $f(x_1:\tau_1, \ldots, x_n:\tau_n): \tau_{n+1}$:= exp;

is type correct if the type of exp is τ_{n+1}, where the type of x_i is τ_i, treated as a constant type.

2. Building up Type Constraints. If an expression contains variables whose type is generic, the simple type evaluation is no longer possible and needs to be modified. In this case, type checking is done by building up constraints on the types of variables. We explain the process by means of an example.

Let us assume that the following declarations have been made:

function append(llist(a),llist(a)): llist(α);
function maplength(llist(llist(α))): llist(integer);

The expression to be type-checked is:

(1) maplength(append(x,y)) = append(maplength(x),maplength(y)).

At the start, the types of x and y, denoted by τ_x and τ_y, are indetermined, or unconstrained:

$\tau_x \geq \alpha_1, \quad \tau_y \geq \alpha_2,$

In order for the expression append(x, y) to be type-correct, τ_x and τ_y both must match llist(a); thus we get the constraints:

$\tau_x = \tau_Y \geq$ llist(α_3).

In the next step, we get a further specialization,

$\alpha_3 \geq$ llist(α_4),

thus

(2) $\tau_x = \tau_y \geq$ llist(llist(α_4)).

and the type assigned to the left hand side expression is llist(integer).

In checking the right hand side of the equation, we can begin with the type constraints (2) since they constitute necessary conditions on any final assignments of types to the variables x and y. The type checking is then trivial: the arguments of append have type llist(integer), which is also the result type. Thus the type constraints (2) lead to the "most general" type assignment

(3) $\tau_x = \tau_y =$ llist(llist(α_4)).

With this type assignment expression (1) is type correct.

Note that, in contrast to symbols, all occurrences of variables in an expression have the same type. A variable must be thought of as bound by a typed universal quantifier, thus all occurrences of the variable name denote the same entity, rather than different instances.

In general it is necessary to reconcile constraints by attempting to construct a unifier. If constraints are not unifiable, a consistent type assignmcnt cannot bc found, thus the expression is not type correct. In building up constraints it does not matter whether one proceeds bottom-up or top-down.

Type Checking of Lemmas. Checking the type of lemmas that contain undeclared variables requires the algorithm described above for building up and unifying type constraints. The kind of type correctness enforced by the algorithm guarantees that instantiation of lemmas will be sound. For example, if expression (1) is used as a rewrite rule, an instantiation of the left hand side will automatically have a tvpe that is an instance of the most general type assigned to it during type checking.

3. Type Checking and Overloading. The restrictions imposed on overloading guarantee that type ckecking is possible in the presence of overloaded symbols. We extend the notion of type correctness in the following way: Assume that f is an overloaded symbol. A expression $f(e_1, \ldots, e_2)$ is considered type correct if it is type correct in the sense of Section 4.1 for at least *one* of the declarations of f. If it is type correct for more than one declaration, we require the context in which the expression appears to resolve the type of f. For axioms and lemmas any ambiguity must have been resolved, if necessary by appropriate declaration of variables, as discussed in Section 3.5.

In assertions, all variables have constant type, thus the restrictions on overloading will identify the type of a function f uniquely.

5. COMBINING SPECIFICATION AND PROGRAMMING LANGUAGES

This section discusses some aspects of using the specification language together with a programming language, in particular the programming language Ada.

In the introduction we pointed out why and in what way a specification language differs from a programming language. This section is concerned with reconciling the difference so that both languages can be used concurrently.

5.1 ASSERTIONS

The paradigm we use is that of programs augmented by assertions written in the specification language; this approach is most common in program verification (cf. [5]).

Assertions are arbitrary quantifier-free first-order formulae. Assertions describe states of computations in the program. They refer to program objects through the name of program variables and express properties of their values in terms of specification concepts. Variables are the only program entities that can be used in assertions; assertions may not refer to subprograms. Assertions thus represent a mixture of specification and programming language. In order for assertions to be meaningful, program variables have to be. interpreted in such a way that functions of the specification language can be applied to them; in particular, the types of program variables have to be interpreted within the type system of the specification language (cf. the following section).

All functions used in assertions must be introduced first in a specification theory. Assertions can mention any function and constant that is visible. A symbol becomes visible if the theory 'which contains its declaration is included in the program via an Ada-like *with-clause*; for instance,

```
with theory Arrays ;
```

Type correctness of assertuions can be checked by type evaluation, since the type of all program variables is constant. An assertion is type correct iff in the process of type evaluation no type conflict has been detected and the type assigned to the formula is boo 1 e an. Note that possible conflicts resulting from overloaded specification symbols will be resolved by the constant types of variables. For assertions within generic units, a generic formal type can be treated as a constant type (i.e. as a type that cannot be instantiated implicitly).

5.2 SPECIFICATION LANGUAGETYPES AND PROGRAMMING LANGUAGETYPES

The type system for specification languages described here has been designed so as to provide a reasonable degree of compatibility with the type systems of languages like Pascal and Ada. For the reasons given in the introduction, it cannot simply be an extension of the type system of those programming languages. However, a certain compatibility is required as a program variable may occur in assertions and thus has to be associated with a type in the specification language that is derived from its programming language type; in other words, programming language types have to bc mapped into specification language types.

We discuss here in some detail how the types of Ada are mapped into specification language types; for other languages like Pascal this mapping would be straightforward as their type structures are much simpler. Note that we need not bc concerned with mapping specification language types to programming language types as elements of the scpcification language never appear as parts of program text.

The mapping from Ada types to specification types is based on the following rules:

- An Ada subtype is always mapped into the same specification type as its base type.

  Specification types are idealized versions of program types; for instance, the specification type integer denotes the infinite set whereas all Ada integer types denote a finite range. In specifications, we do not distinguish between different subtypes, e.g., different ranges; all finite integer types are mapped into the same (abstract) infinite type. The information contained in constraints is lost in specification statements: if it needs to be preserved, it must be expressed by other means, for instance by explicit constraint predicates.

- An Ada derived type is mapped into the same specification type as its parent type.

  The purpose of Ada derived types is to express a kind of ownership of values, which is not relevant in the context of reasoning about properties of values.

- Discriminant parts of a record type are mapped into ordinary record components.

- A private type is mapped into a new constant specification type with the same name; a private type in a gencric type definition is mapped into a type variable.

  There are always two views of an Ada private type: outside the defining package, its structure is hidden, but inside that package it acts just like an ordinary type. This dual nature of Ada private types presents a technical problem for their appropriate reprcsentation as specification types. The specification language discussed here does not provide a mechanism for linking the abstract type that represents a private type in an outer scope with its implementation type; a proper treatment of this situation requires consideration of relationships between theories, which is beyond the scope of this paper.

- The mapping preserves types (i.e. is the identity) unless the preceding rules state something else.

  However, so many types that are considered distinct in Ada are mapped into the sarne specification type that hardly any Ada type is preserved as a distinct specification type.

We do not have a satisfactory mapping for Ada variant records. A simple mapping would make all variant fields ordinary record fields and ignore the connection between discriminants and variants. Anothcr solution would require introducing a disjoint union of types. We do not elaborate either of these approaches and rather refer to the discussion in Section 2.3.

Below we give cxamples of mappings for most common Ada types. This mapping assumes that specification types for integers, booleans, generic arrays, records and generic pointcrs have been introduced; Ada real and float types and task typcs are not considered here.

In the following table T ' denotes the specification type associated with the Ada type T.

Ada Type	Specification Type
integer	integer
boolean	boolean
subtype I is integer range m . . n	integer
type J is new T	T'
array (I) of T	array(integer,T')
array (1 . . n) of T	array(integer,T')
array (T 1 range <>) of T2	array(T1',T2')
array (<>) of T	array(α,T')
record a:T1; b:T2; endrecord	record(a:T1',b:T2')
type R(c:T0) is record a:T1; b:T2; endrecord	record(c:T0',a:T1',b:T2')
access T	pointer(T')
type T is private	T
generic type T is private	α

Note. The Ada notion of generic type is much more restricted than the generic specification types: many Ada generic types are mapped into constant specification types.

5.3 AN EXTENDED EXAMPLE

This section discusses Dijkstra's Dutch National Flag (DNF) problem. The task is to rearrange red, white, and blue elements of an array so that in the final array the colors form solid blocks representing the Dutch flag.

We begin by developing the theory necessary for specifying the problem and the program for its solution. We make use of previously presented theories, Permutation and Ordered-Integers; these are standard theories like Arrays or Integers and treated as "library theories". Theory Same contains specification concepts tailored for this problem; in this sense it is the specification theory for the DNF problem.

```
theory Same (α2) :
    extends Permutation, Ordered-Integers;
                -- Theories Permutation, Ordered-Integers as given in Section 3.4..
    type intarr = array (integer,α2);
                -- This type declaration fixes the index type. Note that the theory Same has only
                -- one type parameter.

    definition same (A:intarr; i,j:integer; c:α2): boolean :=
        if j<i then true else A(i)=c ∧ same(A,i+1,j,c);
                -- same(A,i,j,c) is true if all elements of the array between indices
                -- i and j have the same value, c.

    lemma s1: same(A,i,j,c) ∧ A(j+1)=c ⊃ same(A,i,j+1,c);
    lemma s2: i<k ∧ j<k ∧ same(A,k,l,c) ⊃ same(trans(A,i,j),k,l,c);
    lemma s3: 14 ∧ l<j ∧ same(A,k,l,c) ⊃ same(trans(A,i,j),k,l,c);
    lemma s4: i≤j ∧ same(A,i,j,c) ⊃ same(trans(A,i,j+1),i+1,j+1,c);
end Same;
```

We are now ready to give a more precise specification of the DNF problem: *Given an array* A0 *of blue, white and red elements, transform it to an array* A *so that*

```
    1≤I ∧ I≤J ∧ J≤N ∧ perm(A,A0) ∧
    same(A,1,I-1,blue) ∧ same(A,I,J-1,white) ∧ same(A,J,N,red)
```

Below we give the implementation of a procedure which performs this transformation. The program is written in Ada, with assertions added as "formal comments" in the style of [2]. The with-clause imports (i.e. makes visible) the specification theory Same, which is treated like a generic package (in Ada terminology; we also assume that the with-clause implies a "use-clause" for the theory). Note that, in contrast to ordinary Ada packages, the theory Same is not explicitly instantiated before use; rather, each occurrence of an identifier declared in the theory amounts to an independent instantiation.

```
with theory Same ;     --  This makes visible all symbols declared above

N: Integer := . . . .
subtype Index is Integer range 1 . . N;
type Color is (red, white, blue):
type ColorArray is array (Index) of Color;

function Swap (A:ColorArray; I, J: Index) return ColorArray;
        --| return trans(A,I,J);

procedure Dutch (A: in out ColorArray; I,J: out Index) is
        --| out (1≤I ∧ I≤J ∧ J≤N ∧ perm(A,in A) ∧ same(A,1,I-1,blue) ∧
        --|          same(A,I,J-1,white) ∧ same(A,J,N,red) );
    K: Integer;
begin
    I := 1; J := 1; K := N+1;

    loop --| 1≤I ∧ I≤J ∧ J≤K ∧ K≤N+1 ∧ perm(A,in A) ∧
        --| same(A,1,I-1,blue) ∧ same(A,I,J-1,white) ∧ same(A,K,N,red);
        exit when J >= K;
        if A(J)=blue then
            A := Swap(A,I,J);
            J := J+1;
            I := I+1;
        elsif A(J) =white then
            J := J+1;
        else
            K := K-1;
            A := Swap(A,J,K);
        end if;
    end loop;
end Dutch:
```

The assertions given with this program are actually sufficient for automated formal verification; an earlier Pascal version of the same program has been verified using the Stanford Pascal Verifier [5].

Acknowledgments

This work grew out of an attempt at cleaning up the "rule language" used with the Stanford Pascal Verifier [5]. The author wishes to thank the members of the Program Verification Group at Stanford University for helpful comments on earlier versions of this paper.

REFERENCES

[1] *The Ada Programming Language Reference Manual*
US Department of Defense, US Government Printing Office, 1983.
ANSI/MILSTD 1815A Document.

[2] Luckham, D.C., von Henke, F.W., Krieg-Brueckner, B., and Owe, 0.
Anna: A Language for Annotating Ada Programs, Preliminary Reference Manual.
Technical Report, Stanford University, 1984.
forthcoming report.

[3] Luckham, D.C. & Suzuki, N.
Verification of Array, Record and Pointer Operations in Pascal.
ACM Transactions on Programming Languages and Systems 1(2):226-244, Oct., 1979.

[4] Milner, R.
A Theory of Type Polymorphism in Programming.
Journal of Computer and Systems Sciences 17(3):348 -375, 1978.

[5] Stanford Verification Group.
Stanford Pascal Verifier User Manual.
Report STAN-CS-79-731, Computer Science Department, Stanford University, March, 1979.

[6] Solomon, M.
Type Definitions with Parameters.
In *Proc. of the 5th Symposium on Principles of Programming Languages,* pages 31-38. ACM, 1978.

APPENDIX’

APPENDIX A: SYNTAX

The syntax given below uses the following notation: Syntactic categories (non-terminals) are denoted by names enclosed in "<. .>". In productions, alternatives are separated by " | "; optional parts are enclosed in "[ ]"; optional parts that may be repeated zero or more times are enclosed in "{ }". Names are identifiers. Syntactic categories named “<something – list>" are defined as

```
<something-list>     ← <something> {, <something-list>}


<theory>             ← theory <theory-name> [<theory_params>] :
                           [ extends <theory-instance-list>; ]
                           <theory_body>
                       end [<theory_name>];
<theory_params>      ← (<type_var_list>)
<theory-instance>    ← <theory-name> [(<type_list>)]
<theory_body>        ← <theory_item>; {<theory_item>}
<theory-item>        ← <declaration> | <definition> | <axiom-or-lemma>
<declaration>        ← <type_decl> | <function_decl> | <constant_decl> | <variable_decl>
<type_decl>          ← type <type_name> [(<type_var_list>)] [= <type>]
<type>               ← <type_var> | <enumeration> | <type_name> [(<type_list>)]
<type_var>           ← α | al | a2 | ...
<enumeration>        ← (<ident_list>)
<function_decl>      ← function <function-name> (<type-list>): <type>
<constant_decl>      ← constant <constant-name> : <type>
<variable_decl>      ← var <variable_list> : <type>
<definition>         ← definition [<definition-name> : ] <function_def_list>
<function_def>       ← <function-name> (<parameter-list>): <type> := <expression>
<parameter>          ← <variable_list> : ‘<type>
<axiom_or_lemma>     ← <keyword> <name> : <expression>
<keyword>            ← axiom | lemma
<expression>         ← <cond_expr> | <boolean_expr>
<cond_expr>          ← if <expression> then <expression> else <expression>
<boolean_expr>       ← <term> | ¬ <term> | <term> <boolean_op> <boolean_expr>
<boolean_op>         ← ∧ | ∨ | ⊃ | ≡
<term>               ← <simple_term> | (<expression>) | <simple_term> <relation> <term>
<relation>           ← = | ≠ | ...
<simple_term>        ← <variable> | <function-name> (<expression-list>)
<variable>           ← <variable-name>
```

APPENDIX B: A THEORY FOR POINTER STRUCTURES

The following presents a theory for elementary pointer structures and operations. It attempts to provide a more precise axiomatization of pointer operations than commonly given (see, e.g. [3]).

For an arbitrary type α, the theory introduces two new types denoting *collections* (or "reference classes") of allocated objects of type α and *pointers* to those objects. In order to avoid having to deal with non-determinism we make the simplifying assumption that in any given state of the collection allocation of a new object is *uniquely* determined, so that allocation can be described by *functions* on collections. The new object to be allocated next is the value of the function `new` when applied to a collection; the function `extend` maps a collection C into the extended collection that. includes the object `new(C)`. No assumptions are made about the domain of allocatable objects, except that it contains an unbounded number of elements.

The remaining functions of the theory are for storing into and reading from referenced objects (`store` and `select`), and for testing whether the value of a pointer denotes an allocated object (`allocated`) and whether a value has been assigned to that object (`assigned`). The functions `allocated` and `assigned` are used in the axioms as "enabling predicates" for storing and selecting: the axioms do not say anything about the meaning of `store` and `select` in the case where the premises are false. By adding these premises one can avoid having to deal with potential undefinedness of `store` and `select`.

The Theory for Pointers.

```
theory Pointers (α):
    type collection(α);
    type pointer(α);

    function extend   (collection(α)): collection(α);
    function new      (collection(α)): pointer(α));
    function store    (collection(α),pointer(α),α): collection(α);
    function select   (collection(α),pointer(α)): α;
    function allocated (pointer(α),collection(α)): boolean:
   'function assigned  (pointer(α),collection(α)): boolean:
    constant null: pointer(α);

    var C   : collection(α):
    var p,q: pointer(α);   .
    var e   : α;

    axiom P1 :  ¬ allocated(null,C);
    axiom P2 :  ¬ allocated(new(C),C);
    axiom P3:   allocated(new(C),extend(C));
    axiom P4 :  allocated(p,C)  ⊃  allocated(p,extend(C));
    axiom P5:   allocated(p,C) ∧ allocated (q,C)
                ⊃  allocated(p,store(C,q,e));
    axiom P6 :  allocated(p,C)  ⊃  assigned(p,store(C,p,e));
    axiom P7 :  assigned(p,C)  ⊃  allocated(p,C);
    axiom P8 :  assigned(p,C)  ⊃  assigned(p,extend(C));
```

```
        axiom P9 :  assigned(p,C) ∧ allocated (q,C)
                    ⊃ assigned(p,store(C,q,e));
        axiom P10:  assigned(p,C) ⊃ select(extend(C),p)=select(C,p);
        axiom P11:  allocated(p,C)
                    ⊃ extend(store(C,p,e))=store(extend(C),p,e);
        axiom P12:  allocated(p,C) ⊃ select(store(C,p,e),p) = e;
        axiom P13:  allocated(p,C) ∧ q≠p
                    ⊃ select(store(C,p,e),q) = select(C,q);
    end Pointers;
```

The type pointer(a) may be used in the definition of recursive types (cf. Section 2.2); for instance, in a context where the theory Pointers is visible a type of labelled binary trees using pointers may be defined by,

```
    type btpointer = pointer(bintree)
    type bintree   = record (label: T; left,right: btpointer);
```

As mentioned in Section 2.2, the type name bintree may be used in the declaration of type btpointer before its own declaration.