Partial Implementations
in
Program Derivation

Jan F. Prins
Ph.D. Thesis

87-854
August 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501
PARTIAL IMPLEMENTATIONS
IN
PROGRAM DERIVATION

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Jan F. Prins
August 1987
Abstract

A partial implementation of an abstract notation provides an implementation for some of the notation's operations on some of the values in the notation. A collection of partial implementations of a fixed notation—differing in the selection of values and operations implemented—caters to different patterns of usage of the notation in individual programs. Partial implementations of a general mathematical notation are more appropriate to the formal development of programs than the more familiar paradigm of abstract data types with complete implementations. Furthermore, partial implementations provide the only realistic account of the implementation on finite machinery of many familiar mathematical notations. From a practical point of view, partial implementations of a fixed notation exhibit great reusability and provide a convenient approach to early prototyping in program development.

The incorporation of mathematical notations into a programming system is studied, with particular regard to the formal development of programs in the style of Dijkstra and Gries. A new notion of encapsulation is presented to define partial implementations and a predicate-transformer characterization of implementation correctness is defined. The implementation correctness criterion simplifies and extends the original data-type implementation criterion of Hoare, generalizes implementation criteria for algebraic abstract data types, and accommodates the implementation of non-deterministic operations. Different variables of the same type may have different representations and implementations of operations within the same program. Not all implementations are adequate in a given program, so syntactic and semantic conditions are given to ensure that a proposed implementation of a variable is adequate.
Biographical Sketch

Jan F. Prins was born on August 24, 1956 in Ithaca, N.Y. to Dutch parents during the course of his father’s post-doctoral studies. Transplanted to Holland several years later, he attended a Montessori school where his main interests appear to have involved tracing train tracks throughout the world atlas. In 1968 his family, numbering six with three sisters, returned to the United States to take up residence in Syracuse, N.Y.

He attended Syracuse University and received a B. Sc. with Honors in Mathematics in 1978 while working as a systems programmer for the University Computer Center. He spent one year abroad as a student at the Technische Hogeschool in Eindhoven in Holland, where he received an introduction to the formal derivation of programs taught by E. W. Dijkstra.

After several years of building APL interpreters for a large timesharing firm in Washington, D.C., he enrolled as a graduate student in the Computer Science Department at Cornell University, where he received the M. Sc. degree in 1983. He was fortunate to be able to join his advisor, David Gries, on the latter’s sabbatic year with the Programming Research Group in Oxford. During this year he met his wife Jane in the Royal Oak Pub.

He spent the 1986-87 year as a Research Associate to Tom Reps at the University of Wisconsin at Madison and is currently an Assistant Professor of Computer Science at the University of North Carolina at Chapel Hill.
Acknowledgements

Many thanks are due David Gries, Robert Constable and Robert Capranica, members of my special committee, for their interest and advice. David Gries has worked with me very patiently for many years during which I've learned far more from him than is recorded in this thesis. I hope that I can bring some of the same care and interest to the field that he has shown me.

Over the years I have received plenty of technical assistance, and it is a pleasure to acknowledge the help of Dean Jacobs, Tom Reps, Fred Schneider, Bowen Alpern, David Rossiter, Bob Harper, Rob McCurley, Steve Bartels and Phil Pfeiffer.

Friends and family have all shown tremendous support throughout this undertaking. Thanks to Julie and Rob for putting me up, and thanks to everyone for putting up with me—it took quite a while. Special thanks are due to members of my extended family and especially Jane. Jane never once lost her sense of “humour” about the production of a thesis, and that has been the biggest help of all.

Finally, it is an honor for me to dedicate this work to the memory of my father Willem Prins and his father Jan Albert Prins; two scientists, scholars, and humanitarians who have been an inspiration to me all my life.
Table of Contents

1. Introduction ........................................................................................................... 1
   1.1. Programming Notations .............................................................................. 1
   1.2. Implementation of Abstract Notations ....................................................... 2
   1.3. Adequacy of Partial Implementations ........................................................ 3
   1.4. Efficient Partial Implementations ............................................................... 3
   1.5. Program Correctness and Methodological Considerations ......................... 4
   1.6. Implementation Modules and Directives ..................................................... 6
   1.7. Outline of the Thesis .................................................................................... 7
   1.8. Expository Notation .................................................................................... 8

2. Programming Notation .......................................................................................... 11
   2.1. Overview ...................................................................................................... 11
   2.2. Notation for the Derivation of Programs .................................................... 11
       2.2.1. Basic components of the notation ......................................................... 12
       2.2.2. Variables ............................................................................................. 12
       2.2.3. Scopes ................................................................................................. 13
       2.2.4. Expressions ......................................................................................... 13
       2.2.5. Statements ......................................................................................... 14
       2.2.6. Identifier visibility rules and syntactic restrictions ............................... 14
   2.3. Programming Logic ....................................................................................... 16
       2.3.1. Initial definitions of the predicate transformer $wp$ .............................. 18
       2.3.2. Some properties of scopes and statements ........................................... 23
   2.4. Semantics .................................................................................................... 24
       2.4.1. A relational model for programs .......................................................... 24
       2.4.2. Consistency of the programming logic and relational model ................ 27
       2.4.3. Modeling specifications ....................................................................... 27
       2.4.4. Modeling implementations ................................................................. 28

3. A Type System ...................................................................................................... 30
   3.1. Types of Values in a Programming Notation .............................................. 30
       3.1.1. Type definitions .................................................................................. 33
       3.1.2. Parameterized type definitions .......................................................... 36
   3.2. Type Inference System ................................................................................ 37
List of Figures

Fig. 1 Type definition $Bool$ .................................................................................................................. 35
Fig. 2 Type definition $Set(\sigma)$ ......................................................................................................... 37
Fig. 3 BITV implementation module .................................................................................................... 47
Fig. 4 General form of an implementation module .............................................................................. 48
Fig. 5 P_INT implementation module ................................................................................................. 51
Fig. 6 P_STMT implementation module .............................................................................................. 52
Fig. 7 INT_ARRAY implementation signature ...................................................................................... 57
Fig. 8 Generic ARRAY implementation signature ................................................................................ 59
Fig. 9 Implementation TWO with coercion to and from INT ............................................................... 60
Fig. 10 HEAP module without representation decisions ....................................................................... 65
Fig. 11 Generic HEAP implementation module .................................................................................... 67
Fig. 12 Type definition $Bool$ .............................................................................................................. 82
Fig. 13 Type definition $Set(\sigma)$ ....................................................................................................... 83
Fig. 14 Type definition $Seq(\sigma)$ ..................................................................................................... 84
1 Introduction

1.1. Programming Notations

In the process of programming, a description of a problem to be solved must be turned into machine operations that will solve it. The problem description, or specification, is a highly formalized account of the problem that is most reliably given in a familiar notation. It should be clear from examination of technical literature that precision in meaning is achieved with mathematical notations.

A mathematical notation is natural because it models the problem domain and abstract because it reflects only the salient characteristics and properties of that domain. Properties of a domain are given as a set of identities or relations between terms of the notation and form the basis for formal manipulation. For example, commutativity and associativity of addition are a property of the integers that allows us to rearrange the terms in an addition as we wish without altering the meaning of the whole. Because of their precision and suitability to formal manipulation, mathematical notations are the specification language of choice.

A machine-oriented or operational notation, on the other hand, models the machine; its values can be directly represented within the machine, and its operations are readily (efficiently) computable by the machine. Since machinery is of finite capacity and stylized in construction many natural mathematical notations are not fully (e.g. integers) or efficiently (e.g. sets) implementable. Consequently there is a gap between mathematical and operational notations that must be bridged in the process of programming.

A variety of approaches to the problem exist. A compromise measure, exemplified by programming languages like SETL and APL, abandons the efficiency of low-level operational notations to provide a high-level language that is "closer" to a particular mathematical notation (e.g. set theory). But underlying these languages is a commitment to mechanical execution: the inability to write expressions that denote infinite sets or arrays, or the absence of certain "expensive" primitives betrays this. Consequently, not every "natural" statement of a problem is part of the language.

A more recent example of a high-level programming notation is the PROLOG language. Here the statement of the problem in a subset (Horn-clauses) of first order predicate logic constitutes a directly executable program. But there are many simple formulations of problems that are not immediately in this restricted form, and hence are not directly expressible in the notation. PROLOG is further complicated by the fact that its semantics are not those of first-order predicate logic, but depend on the theorem prover used in the system. This is also a concession to efficiency. It is important not to belittle these concerns, since in practice they make the difference between a feasible and an infeasible solution to a programming problem, but they interfere with the generality and simplicity of a mathematical notation. Although the objectives of high-level programming languages are laudable, their realizations fall short of their mark.
A different approach to the unification of the mathematical and operational notation is to "build" a mathematical notation out of the operational notation, using an encapsulation mechanism (variously called a module, package or class) that associates operations (named to suggest their mathematical meaning) with their implementations. Although such encapsulations can be made very efficient, they are really not abstract. Their operational construction "shines through" the operations they implement: they can not fully model some properties of their mathematical objective. Essentially, their semantics are determined by the details of their implementation.

A more careful treatment introduces notation abstractly —free of representation— by presenting only properties of the notation. This is the approach pioneered for data type definitions by Gutttag [Gutttag75] and further formalized and elaborated by many. The properties are given as a collection of equations (following a restricted form) between operations of the notation.

We shall follow the spirit of this approach in the definition of new notation, but we will not limit the form of the equations that express the properties of the notation. We want to specify a larger class of notations than the (first-order) algebraic methods; for example, we would like to describe higher-order functions and give induction rules. The generalization also simplifies the introduction of partial functions, which occur quite frequently and naturally in mathematical notations. On the other hand, with the elimination of these restrictions the existence of a simple model of our specification is not assured, and questions of consistency or completeness of our equations become even harder. By keeping to familiar mathematical notations, however, we are usually assured of a "standard" model for a set of equations that is well-known and for which the problems of consistency and completeness have been studied.

1.2. Implementation of Abstract Notations

The introduction of abstract notation places us clearly on the mathematical side of the programming divide, and some notion of implementation is needed if programs are to be produced as the end-product of the programming process. A standard view [ADJ78] takes as implementation any (operational) notation that satisfies at least all the identities of the specification. But this returns us to our original problem: an implementation must exhibit all the behavior of a specification, and we have seen that complete mathematical notations can not be fully implemented. This approach excludes these mathematical notations from use in programming.

In order to implement mathematical notations we will need a different notion of implementation; one that does not require complete realization of the notation. We investigate implementations of some values and some operations of a mathematical notation. A partial implementation identifies (by syntactic and semantic requirements) some restriction of a general notation and couples the restricted notation to an implementation. The implementation can be given with a different (operational) notation or with the same notation (typically) used in a more restricted fashion. Certain partial implementations can be considered "supplied"; they define restrictions of notations that couple directly to machine-level operations. A variety of partial implementations of a given notation —differing in efficiency and completeness of operations implemented— cater to differing usage of the same notation in programs.

Another approach to implementation of abstract notation is essentially theorem proving. Term-rewriting systems like RAP [Hussman85], REVE [Lescanne83] and FASE [Kamin83b]
apply all operations symbolically and attempt, when necessary, to simplify an expression to a ground term using the equations. By lifting restrictions on the form of the equations we may seem to make it more difficult to develop a system that implements the notation symbolically. But again, by keeping to familiar notations, we hope that a generalized term-rewriting system, which is highly inefficient, can be replaced with one targeted specifically to the notation (e.g. set theory). There has been considerable success with a generalized proof system that can be adapted with "tactics" to build powerful specialized theorem provers [Constable86]. Moreover, we could always give a "first-order" restriction of a general notation that could be implemented with a term-rewriting system. We therefore take the view that a theorem prover based on the equational theory of a notation is a particular sort of partial implementation.

Partial implementations allow us to program with true mathematical notations and yet obtain a final product suitable for machine execution; they bridge the programming divide.

1.3. Adequacy of Partial Implementations

A complete implementation $C$ of an abstract notation $A$ has the property that any program that uses the operations of $C$ in place of the operations of $A$ will preserve overall behavior; $C$ is adequate for any client of $A$. Partial implementations, on the other hand, are only adequate for some client programs and then only in some conditions. A mathematical notation with an infinite domain of values, for example, can have no implementation that is adequate for all clients. Therefore, in the formal development of a program, the use of a particular partial implementation will require a proof of adequacy. Fortunately, in many cases it is possible to make this proof very simple or even reduce it to verifying syntactic properties of the program and implementation. We introduce weak implementations to simplify adequacy proofs at the cost of possible execution-time errors.

A familiar example of a weak implementation is the arithmetic unit of a 32-bit wide processor operating on 32-bit binary representations of integers. The arithmetic unit is a partial implementation of the integers since it models the abstract operations on a limited domain. As long as our program operates on values in that domain, the implementation will be adequate. But since the arithmetic unit will generate an (overflow) exception on any operation that yields a result outside of that domain, it can also be considered a complete but weak implementation of the full set of integers. A programmer willing to receive "overflow" as an answer can elect to use machine arithmetic without further examination of the adequacy of the implementation in his problem.

1.4. Efficient Partial Implementations

Machine arithmetic is a partial implementation of the abstract integers and is attractive precisely because it has a very efficient realization. The opportunity for efficient partial implementations becomes more evident with notations defined on "aggregate" values. Consider, for example, a comprehensive mathematical set notation. Such a notation makes no restriction on the size of a set or the type of its elements, and includes the full spectrum of set operations such as set union, application of a function to each element in the set, the subset relation and so forth. A full implementation of the notation must be prepared to execute any of these operations, and this presents the implementer with a difficult choice: a representation of a set that is efficient for one operation might not be efficient for another. Which operations should be the most efficient?
The answer depends on the particular pattern of usage of set operations, and will vary from program to program. A full implementation that attempts to make all operations efficient independent of their pattern of usage will certainly be less efficient than a partial implementation "customized" to the given situation. This is a familiar problem for complete implementations of high-level notations with a rich set of primitive operations and leads to compromises in both generality of notation and efficiency of implementation.

Another source of inefficiency is due to the functional nature of mathematical notations. A functional notation permits operations to be composed to form larger expressions. An expression is evaluated by applying the proper operation to the values of its subexpressions. The naive application of this rule can lead to an expensive proliferation of temporary values. When values are small and operations correspond to machine instructions, the process of evaluation is simple and efficient, but with larger aggregate values and complex operations the intermediate values created by each operation incur considerable storage and storage-management costs.

In a language that allows explicit manipulation of storage, in the form of variables, a programmer can compose programs in storage-efficient steps, in which the use of storage is more prescribed. But even a small step can still be prohibitive in cost: consider a set operation like \( s := s \cup \{8\} \). The rules for evaluation call for the construction of a set containing the element 8 that participates in a union operation with a copy of \( s \). The resulting set can then be reassigned to \( s \). This involves the creation of two values the size of \( s \), while two values of this size are discarded. A far preferable method of execution would be to update the value in \( s \) directly. A partial implementation can implement an update operation like \( s := s \cup \{8\} \) without having to provide an implementation for the value of the expression \( (s \cup \{\}) \) or even more general expressions like \( (s \cup r) \).

Another approach to the evaluation of large expressions is based on "shared-values". The LISP S-expr strategy is the most famous example of this approach. But in LISP the only operations available are those that are efficient with respect to the shared-value strategy. A partial implementation that implements a subset of a (sequence) notation corresponding to LISP operations can use the same strategy without requiring us to develop our program in LISP.

In short, we expect to be able to single out restricted patterns of usage in general notations that correspond to efficient partial implementations.

1.5. Program Correctness and Methodological Considerations

We have argued that a partial implementation and a proof of adequacy permit us to achieve efficient programs without sacrificing the specification-level notation in which the programs are developed. How does this compare with the definition of appropriate abstract data types (ADT) that have complete implementations for which no proof of adequacy is required?

A partial implementation of one abstract data type may be a complete implementation of another "smaller" data type. For example, a stack is a partial implementation of a sequence that implements only prepending, deletion and examination of the leading element of the sequence. This same implementation is a complete implementation of a stack abstract data type, providing the familiar operations push, pop, mkempty, etc. In a program for which our partial implementation is adequate, we might as easily have used the stack operations in place
of the sequence operations, at which point any implementation of stack would suffice without further checks. In fact, an ADT advocate would argue that our sequence notation is a poor choice to use in developing the program because it requires too much from an implementation!

The reason we favor the partial implementation approach is methodological and concerns the program development process: program development is fundamentally dependent on the properties of the programming notation whether we develop our programs informally or completely formally. It should be clear in the informal setting; we know, for example, that the program \( z := b + c; z := a + z \) will set \( z \) to be the sum \( a + b + c \). The order of addition is unimportant because of the associativity of addition for integers. In the formal program derivation discipline that we will be using (the predicate transformer approach of [Dijkstra76] and [Gries82]), progress in the derivation can be made only by the transformation of predicates by programming language statements or by establishing implications between predicates. The latter are precisely where the properties of the notation are used formally.

Each abstract data type defines a theory (properties of the operations of the type) that is used in the derivation of programs over the operations of the type. Since program derivation is still a largely human endeavor, each new data type requires —formally or informally— mastery of a new theory. The more abstract data types there are, the more such theories have to be learned. There is an essential stability introduced by fixing a notation for an area and developing all programs along the lines of the associated theory: it is easier to become facile with one theory than many.

We have already mentioned that theories for which there exist complete implementations are not as convenient for program development or else may preclude the development of efficient programs. The theory of machine integers does not have a universal associativity of addition property, for example. So not only are there more theories for the various ADTs, but they are more difficult to use as well.

The most difficult problem facing the careful programmer, however, is the choice of a particular data type in the development of a program. Which operations will be required? The ADT programmer must predict this correctly. For if, at some stage in the development, a different set of operations is adopted, the corresponding theory may be different and the completed portion of the development could be invalidated. A fixed specification-level notation provides us with a single domain of discourse for the development of programs that does not change with the particular choice of operations used in the program. Essentially, the ADT approach, we feel, still makes representation commitments too early in the process of program development in an effort to achieve efficient programs. Partial implementations allow efficiency to be a matter of refinement with a far smaller impact on program correctness arguments.

For example, suppose that as part of some program derivation we have two sequence-valued variables \( s \) and \( t \) and the statements

\[
\begin{align*}
s &:= s \cdot t; \\
t &:= []
\end{align*}
\]

(append \( t \) to \( s \)) \hspace{1cm} (1.1)

(make \( t \) the empty sequence)

A general implementation of sequences could be used with \( s \) and \( t \) that includes the sequence append and assignment operations. A better implementation could be used for \( s \) that only supports an update form of append. Or we might choose to refine (1.1) to obtain
\textbf{inv} \quad s_{in}.t_{in} = s^*t \quad (1.2)

\textbf{do} \quad t \neq [] \rightarrow v := t.(0); \quad \text{(head of } t) \\
\quad t := t.(1..); \quad \text{(tail of } t) \\
\quad s := s \uparrow [v] \quad \text{(single element append)} \\
\textbf{od}

With this refinement we could use a linked-list implementation for \( t \) that need only provide access to the head of \( t \), remove the head of \( t \) and be able to test whether \( t \) is empty. We need only be able to perform single element append-to-right operations on \( s \). These two implementations might lead to a more efficient program.

Or we might develop a specialized implementation of two sequence variables \textit{together} that implements the statement

\[ s, t := s^*t, [] \quad (1.3) \]

directly with a constant time rotation of pointers to the head and tail of the linked lists representing \( s \) and \( t \).

In any case, our changes to use the different implementations never changed the abstract semantics of the statement (note that (1.1), (1.2) and (1.3) all have the same meaning), and it was not necessary to change the semantics of sequences in the correctness argument when we used different implementations: \( s_{in}.t_{in} = s^*t \) remains invariant in (1.2) despite the fact that general sequence append might not be an operation supported by the implementations for \( s \) and \( t \).

We have used the word ""notation"" throughout rather than ""abstract data type"" because the definitions for a notation are relatively fixed, mathematical and targeted for the subject area of programming problems, while abstract data type definitions are more problem-specific, characterizing a class of complete implementations. We also have in mind that a notation is thoroughly supported at a syntactic level in the programming process, providing familiar display forms, precedence rules, and so forth.

\subsection*{1.6. Implementation Modules and Directives}

An implementation and a client are independent until the use of the former is directed in the latter. To this end, an

- \textit{implementation module} presents a subset of terms and statements of a notation and couples each to a specialized implementation, and an

- \textit{implementation directive} specifies the implementation module that is to provide implementations for terms involving designated variables within a scope.

An implementation module may be developed independent of client programs and is not executable. Within it, high-level operations are paired with their implementations, possibly drawn from a lower-level programming notation. Each module postulates a \textit{coupling}, an invariant relation between the abstract and implementation values. The coupling is the basis for correctness of the module and must be \textit{preserved} by each set of operations paired in the module.

An implementation directive issued for a variable in a client program is not directly valid. The module provides some syntactic control over the applicability of a module to a client program text. All terms involving the variable in the scope of the directive must be implemented by the module, and preconditions for proper implementation must be achieved by the
corresponding preconditions of the client. A full guarantee that an implementation directive specifies an adequate implementation for the client program may require, in addition to the syntactic checks, simple proofs in the predicate calculus.

Currently, we believe, few programmers use the wealth of data structuring techniques discussed in textbooks, such as AVL trees or specially linked lists. Implementations for them are not very (re-)usable because to use them properly requires the programmer to extract their "meaning" from the procedure names and possible documentation and then develop their program in terms of the procedures provided. In our point of view the implementation modules do not "provide" anything new; they are simply partial implementations of a single notation. A programmer using the partial implementation need only refine the program to use the subset of operations implemented, and can do so without consulting the details of the implementation module. With this view of implementations it should be possible to build modules for important data structures that will be usable by many instead of just their authors.

1.7. Outline of the Thesis

In this thesis we will consider the definition of general programming notations, their partial implementations, and their use in the development of correct programs. This thesis forms a part of a larger objective to develop a programming language, an evolving library of partial implementations, and a system to aid in the development of programs, providing empirical assessments of the approach advocated here. Consequently material is included here to address some of the practical problems that might arise in the construction of such a language, library and system.

Much of the usability of a mathematical notation, we believe, lies in its concise written form. We would like our programming notations to share this feature, so that, unlike an abstract data type that introduces operations as a collection of named functions, new notation introduced to address a new problem domain should be accorded the same status and concise display as conventional "built-in" notation. Chapter 2 considers the basic components of a programming notation and the incremental extension of the syntax and semantics to include new notations to achieve the compact and familiar display forms of mathematical notations. This chapter also introduces the semantic basis for program development based on predicate transformers and places the core components of the notation in that framework. Finally a model is described that can be used to describe the formal foundations of program development. Specifications, programs and partial implementations can all be represented in this model.

In further support of our desire to use concise mathematical notations in program development, chapter 3 investigates a type system in which types can be inferred from a fragment of notation, rather than having to be declared. Since mathematical notations are often generic, that is, invariant under change of types of values on which they operate, the type system must be able to accommodate such notations. Set operations, for example, are generic because they are written the same way — and have the same properties — whether they contain integers, tuples, or whatever values. For this reason we adapt the polymorphic type inference system of Milner [Milner78] to our setting. We consider how the new types and operations of a notation can be included in the type system and how the type inference algorithm can be adapted to extend over the new notation.
One of the implicit functions of "type" in a programming language is to select the correct implementation during compilation. Since we have separated the implementation of a notation from its definition, we find that we can introduce notation with a more general typing than would be the case with the ML language, the language in which Milner's type inference system is employed.

Chapter 4 describes implementation modules and directives and presents some examples of their use. A general facility to describe fragments of the programming notation is described that is used in implementation modules to delineate the portion of the notation implemented and to constrain the interaction of the implementation with cooperating and independent modules. Three approaches to increasing the flexibility of modules are described and the relation to module development is discussed.

Implementation directives are defined and the syntactic determination of their suitability within a client program is outlined. An attractive possibility in the development of implementation modules is the inclusion of implementation directives within their implementation components. An extended example, the development of a generic HEAP module, details the technique.

Chapter 5 describes the role of implementation modules and directives in formal proofs of program correctness. Implementation modules can be shown correct independent of client programs with a correctness criterion that generalizes and simplifies the original correctness rule of Hoare [Hoare72]. The rule extends to give a correct account of the implementation of nondeterministic operations. The rule is insensitive to implementation bias [Jones80], which is precisely what partial implementations exhibit.

A partial implementation that is correct in isolation must still be shown to be adequate for a given client. The verification conditions which ensure adequacy are presented and the notion of adequacy with respect to a specification is discussed.

Appendix I includes definitions and explanations for notations that we have been using to develop programs.

The remainder of this introductory chapter briefly, and informally, explains some of the notation used in the sequel.

1.8. Expository Notation

If we pick a domain of discourse, say numbers, we may write expressions over that domain by combining symbols standing for constants, variables and functions with parentheses to delineate arguments. Conceptually, all function application can be thought of in "prefix" form: if $f$ is a function symbol and $x$ and $y$ are variables, then $f.x$ denotes application of unary $f$ to $x$, while in $f.(x,y)$ the symbol $f$ represents a two-position function. For ease of comprehension, application of many familiar functions will be written in an infix form. For example, $+$ is the usual symbol for the two-place addition function, and we write application as $2 + 3$ or $2 + (x + 3)$ in order to delineate the arguments.

A proposition is an expression over the domain of truth values ($true$, $false$), variables, and the familiar connectives $\land, \lor, \neg$, etc. A predicate is a proposition with one or more variables replaced by a truth-valued relation (e.g. $=, \neq, \leq$, etc.) between expressions. For example $a < b \land b < 5$, which we abbreviate as $a < b < 5$. In these examples, occurrences of variables $a$
and \( b \) are \textit{free}; the value of the expression is dependent on the values of these variables. In the case of a predicate \( P \), an assignment of values to the variables that occur free for which \( P \) has the value \textit{true} is said to \textit{satisfy} \( P \). For example, \( a \) and \( b \) occur free in the predicate \( P : a < b \), the assignment \( a = 5, \ b = 10 \) satisfies \( P \), while the assignments \( a = 10, \ b = 5 \) and \( a = b = \text{true} \) do not satisfy \( P \).

In contrast, within the extent of a \textit{scope} \( (v \parallel P \parallel E) \), all occurrences of \( v \) are \textit{bound}. The \textit{domain predicate} \( P \) characterizes the possible values that \( v \) may assume in the evaluation of expression \( E \). When \( v \) is a list of identifiers, all identifiers in the list are simultaneously bound and satisfy \( P \) within the scope. There are a variety of operators over scopes, but in every case the meaning of the final construct is \textit{independent} of the bound variable(s) \( v \).

Universal and existential quantification are operators over a scope \( (v \parallel P \parallel R) \) where \( R \) is a predicate. The predicate \( \forall (v \parallel P \parallel R) \) is satisfied when \textit{every} value of \( v \) that satisfies \( P \) also satisfies \( R \). Similarly \( \exists (v \parallel P \parallel R) \) is satisfied if \textit{some} value of \( v \) that satisfies \( P \) also satisfies \( R \). In both definitions, an assignment of values that satisfies the complete construct does not assign a value to the bound occurrences of \( v \) within the scope.

Certain predicates characterize the values of a given type. That is, they are satisfied by all and only values of the type, such as the predicate \textit{Numeric}. \( v \), which is \textit{true} if \( v \) is a number, but \textit{false} otherwise. It is convenient to distinguish these predicates and to write them separately in a scope. We write

\[(v : \text{Numeric} \parallel P \parallel R)\]

in place of

\[(v \parallel \text{Numeric}. v \land P \parallel R)\]

These predicates can be treated separately and automatically by a type-checking system. Often they can be inferred from the operations used in \( P \) and \( R \).

\textbf{Remark}

In the case of universal quantification, we must be careful about inference of type predicates based on the operations in \( R \). Consider the predicates

\[\exists (v \parallel \text{true} \parallel a + b = b + a) \quad \text{and} \quad \forall (v \parallel \text{true} \parallel a + b = b + a)\]

Assuming that \( + \) is an operation defined only on \textit{Numeric} values, we might infer that \textit{Numeric}(\( v \)) is a safe addition to the domain predicate \textit{true}. But note that under this assumption the value of the universally quantified predicate changes, while the existentially quantified predicate does not. Whereas it was initially "undefined" for some values for \( a \) and \( b \) (choose \( a = b = \text{true} \)), when \( a \) and \( b \) are restricted to \textit{Numeric}, it becomes a tautology, satisfied by all choices for \( a \) and \( b \). In general we are not interested in predicates with undefined values, and so we usually mean the restricted interpretation, but a type inference system should not change the meaning of any expression, so on occasion we will have to supply type predicates ourselves. \( \Box \)

The set \textit{constructor} \textit{set} is defined over a scope with an arbitrary "range" expression \( E \). For convenience we write \( (v \parallel P \parallel E) \) in place of a more cumbersome \textit{set}(\( v \parallel P \parallel E \)). The set consists of the value of \( E \) for each \( v \) that satisfies \( P \). For example, \( \{v : \text{Numeric} \parallel 2 \leq v \leq 5 \parallel v^2 \} \) is the set \( \{4, 9, 16, 25 \} \).
The set reduction operator "\langle\rangle" is defined on a scope \((\nu \parallel P \parallel E)\) and a function \(f\), where \(P\) is a predicate, \(E\) is an expression of some type \(\sigma\), and \(f\) is of type \(\sigma \times \sigma \rightarrow \sigma\) (and is commutative and associative). Function \(f\) is applied between all values of \(E\) with \(\nu\) satisfying \(P\). We write distribution as \(f \langle i \parallel P \parallel E \rangle\). For example,

\[
\begin{align*}
+ \langle i \parallel 1 \leq i \leq n \parallel 2 \cdot i - 1 \rangle \\
= 1 + 3 + \cdots + 2 \cdot n - 1 \\
= n^2
\end{align*}
\]

If \(f\) has an identity \(k\), then we define \(f \langle i \parallel \text{false} \parallel E \rangle \triangleq k\).

There are other operators and constructors over scopes, but since all of them are part of the programming notation and are not immediately used in the following, they will be defined with the programming notation in the next two chapters and in the Appendix.
2 Programming Notation

2.1. Overview

In this chapter we define the foundations on which a programming notation can be built. Section 2.2 introduces the key productions of the grammar that is extended with the introduction of new notation. Variables and their scope are defined over expressions and statements and a "core" notation is given after which further extensions can be uniform in their treatment of identifier visibility. Two entirely new programming constructs are introduced; a specification statement that connects program development in a very direct way with the programming notation, and a refinement construct for expressions that allows expressions to have an internal state without complicating program development.

Section 2.3 introduces the logic used for program development and its relation to new notation. The core notation that was introduced in section 2.2 is placed within the logic.

Section 2.4 introduces a single unified model for the semantics of specification, programming language constructs and implementations. The core notation is reflected in this model and consistency between the model and the logic is discussed.

2.2. Notation for the Derivation of Programs

Much of the notation we expect to use in program derivation is problem-domain specific. Consequently, the values and operations available to us will be introduced by independent type definitions that extend basic definitions common to all notations. Physicists, for example, might define complex numbers, vectors, and matrices, while computer scientists might be more inclined to use integers, sequences, and sets. The basic type definitions provide the machinery to introduce and formalize the use of new notation in program development.

A principal objective of a notation is to simplify program derivation and correctness arguments. We impose a number of requirements that must be met by a notation in order to facilitate, and perhaps partially automate, this process. These are:

- One-level store semantics. This means there are no pointer or "left-hand values" in the notation. Each variable stands directly for a value and never for a name or address of another variable; thus there can be no aliasing between names, and all operations on a variable are manifest.

- Pure expressions. All expressions are free from side effects (they do not change any variables), all functions are referentially transparent (their value is fully determined by their parameters, and not the environment of their call or their "own" state), and all expressions are deterministic (their result is determined uniquely by their free variables).

- Strong typing of program and proof. A fixed portion of correctness proof obligations can be completed syntactically. For any program fragment we can infer the types required of the free occurrences of identifiers and reject type-incorrect programs and proofs.
Although these restrictions are attractive for program correctness arguments, they are usually thought to be too restrictive for the development of efficient programs. For example, pointers to large data values are far less expensive to manipulate in a conventional machine than the data values themselves, but this is precluded by the first item above. We will try to obtain the benefits of both simple program development and efficiency by allowing notations that do not meet the requirements above to appear in implementations. Consequently the development of efficient implementations may be more tedious than the development of pure programs, but this is exactly where such efforts should be expended if we the end product is a library of reusable and efficient implementations.

2.2.1. Basic components of the notation

The basic components of the notation are expressions and statements. The syntactically valid instances of these categories are described by the nonterminals \(\langle exp\rangle\) and \(\langle stmt\rangle\) of a grammar \(G\) that will generate the complete notation. The initial form of the productions of \(G\) are given below; new functions and statements introduced in type definitions add additional right-hand sides to productions for \(\langle exp\rangle\) and \(\langle stmt\rangle\). Productions are given an abstract syntax that uniquely identifies each possible right hand side and the nonterminals within it.

The abstract syntax is really only convenient for standardizing definitions and for uniformity in manipulation. To make the definitions more convenient to use and remember, a rule for display form accompanies each production. These rules direct the appearance of an instance of the production in the printed form of the notation. A simple way to state these rules is as concrete syntax, in parallel with the abstract syntax, although ultimately this will not offer sufficient flexibility. More sophisticated rules, such as operator precedence and parentheses, could be expressed, for example, by attribution of the grammar with functions to build up the context-sensitive display form. Although this sort of approach is required in a usable programming system, it is not undertaken here. Moreover, instead of “fully-parenthesizing” the display form, we will often leave the display form ambiguous and rely on the experience of the reader with traditional mathematical notations to resolve the intended meaning. With the exception of the definitions below, the display form determined by the concrete syntax is used exclusively.

We turn now to the initial definition of the non-terminals of \(G\). Since we will be making a number of definitions, like display form, in parallel with the abstract syntax, it will be convenient to label the individual non-terminals in the abstract syntax for use in the definitions. Thus the label \(E\) in \(\langle exp\ E\rangle\) stands for any expansion of \(\langle exp\rangle\). Later we will augment this convention to restrict the possible expansions of the nonterminal in order to build patterns that match subsets of the sentences in \(G\).

2.2.2. Variables

Variables are introduced into the notation by the non-terminal \(\langle id\rangle\) of \(G\). The set of lower case alphabetic strings (i.e. \(a, a a, \ldots, b, \ldots\)) are fixed as the terminals of \(\langle id\rangle\). The display form of a variable is, as usual, the textual representation of its identifier. For convenience we define a linear list of identifiers \(\langle idl\rangle\), typically used to introduce variables in a scope. More precisely,
abstract syntax | display form
--- | ---
\(\langle idl \rangle ::=\)  
\(\langle id x \rangle\)  
\(\langle tuple \langle id x_i \rangle_{i=0, \ldots, n-1} (n > 1) \rangle\)  
\(x\)  
\(x_0, \ldots, x_{n-1}\)

### 2.2.3. Scopes

Occurrences of a variable are free unless they are bound within a scope (see the discussion in section 1.8). A scope, in itself, has no meaning but may be incorporated within productions of \(G\) to define operators over expressions and statements that have bound variables. A scope is defined as

\[
\langle scope \rangle ::= \langle scope \langle idl \overline{v} \rangle \langle exp P \rangle \langle exp W \rangle \rangle  
\langle \langle scope \langle idl \overline{v} \rangle \langle exp P \rangle \langle stmt W \rangle \rangle \rangle
\]

\(P\) is the domain predicate and \(W\) is an \(\langle exp \rangle\) or a \(\langle stmt \rangle\). Each identifier in \(\overline{v}\) is bound throughout \(P\) and \(W\). Nominally the display form is \((\overline{v} \parallel P \parallel W)\), but this can be adapted for the definition of certain operators (e.g. the set constructor \((\overline{v} \parallel P \parallel W)\)).

### 2.2.4. Expressions

Expressions denote values. The variety of values is determined by the collection of type definitions, but certain values, namely tuple- and function- values, are defined here. The initial definitions for production \(\langle exp \rangle\) are

abstract syntax | display form
--- | ---
\(\langle exp \rangle ::=\)  
\(\langle id v \rangle\)  
\(\langle tuple-cons \langle exp E_i \rangle_{i=0, \ldots, n-1} (n > 1) \rangle\)  
\(\langle tuple-extr_i \langle exp E \rangle \rangle\)  
\(\langle lambda \langle scope \langle idl \overline{v} \rangle \langle exp P \rangle \langle exp E \rangle \rangle \rangle\)  
\(\langle apply \langle exp f \rangle \langle exp E \rangle \rangle\)  
\(\langle valblock \langle scope \langle idl \overline{v} \rangle \langle exp P \rangle \rangle\)  
\(\langle \langle stmt S \rangle \langle exp E \rangle \rangle \rangle\)

\(v\)  
\((E_0, \cdot \cdot \cdot, E_{n-1})\)  
\(E, i\)  
\(\lambda(\overline{v} \parallel P \parallel E)\)  
\(f . E\)  
\((\overline{v} \parallel P \parallel S yields E)\)

In order, these expressions stand for
- The value denoted by a variable, obtained by reference to the identifier.
- A tuple value built out of component values.
- An element extracted from the \(i^{th}\) component of a tuple. The value of \(i\) is really part of the production name and therefore the \(tuple-extr\) production is really a family of productions.
- An expression \(E\) abstracted over its free variables to form a function value with domain \(P\).
- A function applied to a value to yield a new value.
- The value of expression \(E\) following a statement \(S\) (see below) in a scope that limits side effects of \(S\). This definition links statements with values; an expression may have internal
state in its evaluation. This is a mechanism for refinement of an expression.

Remark

An expression may be abstracted over its free variables to yield a function, but we omit at this time abstraction of statements over free variables to yield procedure values, since this complicates the presentation. There is no real technical difficulty, but the proof rules for parameterized definitions whose arguments may vary over procedure values are complex since properties of a procedure parameter must be stated in terms of the pre- and post-condition of the argument procedure. For an account of the work involved see Olderog [Olderog84] or Ernst [Ernst82]. Function-valued arguments, however, present less difficulty because we can specify requirements of a function argument in terms of the function itself; since functions normally appear in assertions, these "second-order" predicates are simpler to manipulate. □

2.2.5. Statements

Statements may alter the association between variables and values. A program is an instance of a statement. The following definitions give the syntax of statements for the assignment of values to variables, the introduction of variables within a scope, the sequential composition of statements, and the achievement of a given relation \( P \) by choosing values for variables \( \nu \).

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>display form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle \text{stmt} \rangle ::= )</td>
<td>( \langle \text{stmt} \rangle ::= )</td>
</tr>
<tr>
<td>( \langle \text{assign idl } \nu \rangle \langle \text{exp } E \rangle )</td>
<td>( \nu := E )</td>
</tr>
<tr>
<td>( \langle \text{block scope idl } \nu \rangle \langle \text{exp } P \rangle \langle \text{stmt } S \rangle )</td>
<td>( (\nu \parallel P \parallel S) )</td>
</tr>
<tr>
<td>( \langle \text{choose stmt } S_i \rangle_{i=0, \ldots, n-1} )</td>
<td>( S_0; \ldots; S_{n-1} )</td>
</tr>
<tr>
<td>( \langle \text{choose idl } \nu \rangle \langle \text{exp } P \rangle )</td>
<td>( \text{choose } \nu \text{ so } P )</td>
</tr>
</tbody>
</table>

Predicate \( P \) in the block \( (\nu \parallel P \parallel S) \) constrains the initial value of \( \nu \) in \( S \). We can use the identity

\[ (\nu \parallel P \parallel S) = (\nu \parallel \text{true} \parallel \text{choose } \nu \text{ so } P ; S) \]  

(2.1)

to uncouple the introduction of \( \nu \) from the satisfaction of \( P \). The form \( (x \parallel \text{true} \parallel S) \) does constrain the initial value of \( x \), but barely so: \( x \) may take on any initial value, since the domain is the predicate \( \text{true} \). It can be seen as a model of a scope introducing a variable without an initial value.

As a consequence of this definition, any scoped variable is bound to some value on introduction. We will make a small leap of faith about the free occurrences of identifiers in a \( \langle \text{stmt} \rangle \) or \( \langle \text{exp} \rangle \) and assume that they are always bound to some value as well. This leads to the simplifying assumption that a variable always denotes a value.

2.2.6. Identifier visibility rules and syntactic restrictions

The definitions above, and those added later, will provide the context-free grammar to parse programs. Now we make some inductive definitions over the productions of \( G \) that determine, for any statement or expression \( W \), the set of identifiers that occur free in \( W \). We will rely on these definitions to extend over whatever notation might be defined by the grammar \( G \) in
order to state further definitions such as semantic requirements that must be met by the expressions and statements in a notation. These definitions can also be used to syntactically restrict the sentences of \( G \) to those that achieve some of the requirements made in section 2.2: the absence of aliasing between distinct identifiers and insuring that functions are dependent only on their arguments and not on any other portion of the state at the point of their call.

An occurrence of an identifier is free unless the occurrence is bound in an enclosing scope. If an expression or statement is analyzed into its components, the \( \langle \text{id} \rangle \) productions introduce free occurrences of variables, while scope productions remove them. For any other production, an occurrence of an identifier is free if it is free in any subproduction. The function \( \text{fv} \) (freely-occurring variables) from abstract syntax to a set of identifiers is defined inductively over the productions of \( G \) as follows.

**Identifiers.** A simple identifier yields a single free occurrence:

\[(a) \quad \text{fv} \cdot \langle \text{id} \ x \rangle = \{x\} \quad (2.2)\]

We extend this definition over \( \langle \text{idl} \ \overline{v} \rangle \) by taking the union of the identifiers in the list:

\[(b) \quad \text{fv} \cdot \langle \text{tuple} \ (\langle x \_0 \rangle \ \cdots \ \langle x \_n \rangle ) \rangle = \{i \mid 0 \leq i < n \| x_i\} \quad (2.2)\]

We will use the shorthand \( [\overline{v}] \) for \( \text{fv} \cdot \langle \text{idl} \ \overline{v} \rangle \).

**Expressions.** If an expression is a scope, we define

\[(c) \quad \text{fv} \cdot \langle \text{scope} \ (\langle \text{idl} \ \overline{v} \rangle \ \langle \text{exp} \ P \rangle \ \langle \text{exp} \ E \rangle ) \rangle = (\text{fv} \cdot P \cup \text{fv} \cdot E) - [\overline{v}] \quad (2.2)\]

For any other right hand side name \( \phi \) in the production \( \langle \text{exp} \rangle \) we define

\[(d) \quad \text{fv} \cdot \langle \phi \ (\langle \text{exp} \ E_0 \rangle \ \cdots \ \langle \text{exp} \ E_{n-1} \rangle ) \rangle = \cup / (i \mid 0 \leq i < n \| \text{fv} \cdot E_i) \quad (2.2)\]

We illustrate these rules on each \( \langle \text{exp} \ E \rangle \) defined thus far.

<table>
<thead>
<tr>
<th>form of ( E )</th>
<th>( \text{fv} \cdot E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle \text{id} \ x \rangle )</td>
<td>( {x} )</td>
</tr>
<tr>
<td>( \langle \text{exp} \ e_0 \rangle, \cdots, \langle \text{exp} \ e_{n-1} \rangle )</td>
<td>( \cup / (i \mid 0 \leq i &lt; n | \text{fv} \cdot e_i) )</td>
</tr>
<tr>
<td>( \langle \text{exp} \ e \rangle \cdot \langle x \rangle )</td>
<td>( \text{fv} \cdot e )</td>
</tr>
<tr>
<td>( \langle \lambda \langle \text{idl} \ \overline{v} \rangle \ \langle \text{exp} \ P \rangle \ \langle \text{exp} \ e \rangle \rangle )</td>
<td>( (\text{fv} \cdot P \cup \text{fv} \cdot e) - [\overline{v}] )</td>
</tr>
<tr>
<td>( \langle \text{exp} \ f \rangle \cdot \langle \text{exp} \ e \rangle )</td>
<td>( \text{fv} \cdot f \cup \text{fv} \cdot e )</td>
</tr>
<tr>
<td>( \langle \langle \text{idl} \ \overline{v} \rangle \ \langle \text{exp} \ P \rangle \ \langle \text{stmt} \ S \rangle \ yields \langle \text{exp} \ e \rangle \rangle )</td>
<td>( (\text{fv} \cdot P \cup \text{fv} \cdot S \cup \text{fv} \cdot e) - [\overline{v}] )</td>
</tr>
</tbody>
</table>

The rule for \( \text{valblock} \) is a generalization of 2.2 (d) that depends on the extension of \( \text{fv} \) over statements, which we consider next.

**Statements.** An identifier that occurs in a statement may be referenced or assigned, and we separate these cases. The referenced identifiers in a \( \langle \text{stmt} \ S \rangle \) are given by \( \text{use}(S) \), the assigned identifiers by \( \text{def}(S) \). Together these two functions define \( \text{fv} \) on statements:

\[(e) \quad \text{fv} \cdot \langle \text{stmt} \ S \rangle = \text{use}.S \cup \text{def}.S \quad (2.2)\]

The definitions of both \( \text{use} \) and \( \text{def} \) follow the same lines as the definition of \( \text{fv} \) over expressions: \( \langle \text{id} \rangle \) productions introduce identifiers, scopes remove them, other productions combine
them. The only additional decision to be made is whether a given identifier is used or defined. We illustrate this decision for each \(\text{stmt} \ S\) introduced thus far:

<table>
<thead>
<tr>
<th>(\text{form of} \ S)</th>
<th>(\text{use.} \ S)</th>
<th>(\text{def.} \ S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle \text{idl} \ \overline{\nu} \rangle := \langle \text{exp} \ E\rangle)</td>
<td>(\text{fv.} \ E)</td>
<td>(\overline{\nu})</td>
</tr>
<tr>
<td>(\langle \text{idl} \ \overline{\nu} \rangle \parallel \langle \text{exp} \ P \rangle \parallel \langle \text{stmt} \ S\rangle)</td>
<td>(\text{fv.} \ P \cup \text{use.} \ S \setminus {\overline{\nu}})</td>
<td>(\text{fv.} \ P \cup \text{def.} \ S \setminus {\overline{\nu}})</td>
</tr>
<tr>
<td>(\langle \text{stmt} \ S_0 \rangle ; \cdots ; \langle \text{stmt} \ S_{n-1} \rangle)</td>
<td>(\cup \langle i \parallel 0 \leq i &lt; n \parallel \text{use.} \ S_i \rangle)</td>
<td>(\cup \langle i \parallel 0 \leq i &lt; n \parallel \text{def.} \ S_i \rangle)</td>
</tr>
<tr>
<td>choose (\langle \text{idl} \ \overline{\nu} \rangle) so (\langle \text{exp} \ P\rangle)</td>
<td>(\text{fv.} \ P)</td>
<td>(\overline{\nu})</td>
</tr>
</tbody>
</table>

Note that we can not hope to be "accurate" in our classification of a variable as being used or defined in a Turing equivalent language, since that is not a decidable property of programs in such a language. All we require here is a conservative assessment: if \(\nu \notin \text{fv.} \ \text{def.} \ S\) then we want to be sure that it is not possible for \(S\) to result in a change to \(\nu\).

Constraints. We can now state three requirements to ensure certain properties of variables and expressions are met that simplify correctness arguments.

First, to prevent aliasing, each of the identifiers in a list of identifiers must be distinct, e.g. \(a, b, a\) is not a sensible target of an assignment or header of a function. Therefore we constrain an \(\langle \text{idl} \ \overline{\nu} \rangle\) of the form \(\langle \text{id} \ x_0 \rangle, \cdots, \langle \text{id} \ x_{n-1} \rangle\) to satisfy

\[
\forall (i, j) \ 0 \leq i < j < n \parallel x_i \neq x_j
\]  

(2.3)

Second, for referential transparency of function values we require that a function is fully abstracted over all its free variables, hence that

\[
\text{fv.} \ \lambda (\langle \text{idl} \ \overline{\nu} \rangle \parallel \langle \text{exp} \ P \rangle \parallel \langle \text{exp} \ e \rangle) \prime = (\text{fv.} \ P \cup \text{fv.} \ e) \setminus \{\overline{\nu}\} = \emptyset
\]  

(2.4)

Finally, to ensure that expressions have no side-effects, we require that statements can alter only local variables in a scope that yields a value, since that scope can be used in an expression. Therefore we constrain

\[
\langle \langle \text{idl} \ \overline{\nu} \rangle \parallel \langle \text{exp} \ P \rangle \parallel \langle \text{stmt} \ S \rangle \ yields \langle \text{exp} \ E \rangle\rangle
\]  

(2.5)

to satisfy \(\text{def.} \ S \subseteq \{\overline{\nu}\}\).

2.3. Programming Logic

It is of great utility to be able to prove that a program achieves an independent "specification" of behavior. We follow the seminal work of Floyd [Floyd67] and Hoare [Hoare69] towards this goal and define a predicate to be a first-order truth-valued expression over program variables. Strictly speaking, we will depend on a three-valued logic (\text{true}, \text{false}, and \text{undefined}) so that predicates define total functions and manipulation of predicates is simplified by obviating the need for non-commutative "short-circuit" propositional connectives like \text{and} and \text{or}. The utility of a logic covering undefinedness in program proofs is discussed in [Barr84, Jacobs85]. Each predicate represents a subset of "program states" consisting of the assignments of values to program variables satisfying the predicate, i.e. for which the predicate has value \text{true}. A \text{specification} is a pair of predicates \((P, Q)\) that are satisfied by acceptable initial and final states, respectively. The \text{precondition} \(P\) and \text{postcondition} \(Q\) fix input-output behavior but do not specify the way in which that behavior is to be achieved.
The verification that program $S$ achieves specification $(P, Q)$ is a proof that $S$ started in any state satisfying $P$ will terminate in a state satisfying $Q$. This proposition is written \( \{P\} S \{Q\} \) (Hoare triple) and, if proved correct, is valid. Hoare [Hoare69] gives a programming logic, a collection of axiomatic triples for the basic statements and rules of inference to compose (in a syntax-directed fashion) ever larger Hoare triples. Thus $(P) S \{Q\}$ is valid exactly when it is a theorem of the programming logic. Sometimes $S$ is viewed as the “proof” that under assumption $P$ there exist values for variables to satisfy $Q$; simultaneously $S$ provides directions in an executable notation to find such values.

The derivation of a program $S$ from a specification $(P, Q)$ is the construction or synthesis of $S$, guided by $P$ and $Q$, in such a way that $(P) S \{Q\}$ is valid. Dijkstra [Dijkstra76] argued that the programming logic for a small “guarded command” language [Dijkstra75] could be stated as a calculus to serve as part of a methodology for program derivation, providing a sound foundation for the correctness of the resultant programs. These techniques were subsequently refined and explained by Gries [Gries81]. Central to program derivation is the predicate transformer $wp.(S, Q)$ (weakest precondition), that yields the least restrictive predicate $P$ such that $(P) S \{Q\}$ is valid. That is, $\{wp.(S, Q)\} S \{Q\}$ is valid, and for any $P$ such that $(P) S \{Q\}$ is valid, $P \Rightarrow wp.(S, Q)$. Hence $(P) S \{Q\}$ is equivalent to $P \Rightarrow wp.(S, Q)$. We adopt this view of the programming logic in our setting.

In our notation, only statements can modify the program state (by altering program variables). We therefore define $wp.(S, R)$ for each $(stmt S)$ and arbitrary predicate $R$. From a syntactic point of view, $R$ is a truth-valued expression, and hence $fv.R$ is the set of variables that occur free\(^1\) in $R$.

In general, we make no a priori distinction between expressions in the programming notation and in predicates (the assertion language), so that the assertion language is always a subset of the programming notation. Of course a reasonable assertion language will contain far more expressions than can be mechanically evaluated, therefore we can not guarantee that all programs in the notation can be executed. Only certain programs and certain expressions will have implementations in a given system. The executable portion of the programming notation varies with the details of implementation, and consequently is determined by an implementation module, i.e. by what a given implementation approach is prepared to provide. This is in contrast to a notation in which some executable interpretation is fixed for all constructs and expressions, thereby segregating the programming and assertion language. For example, in the guarded-command language the class of predicates is larger than the class of truth-valued expressions, since predicates may include quantifiers that cannot appear in the programming notation.

The definition of $wp$ is only a small part of the work necessary to be able to derive and prove programs correct in a notation. The majority of predicate manipulations are justified not by program statements but by identities or implications of the underlying predicate logic. All our predicate logic is axiomatized by identities and implications, but we will not attempt to state all of the properties. By limiting ourselves to “familiar” types and operations (e.g. as

---

\(^1\)Certain variables that appear to occur free in specifications are actually logical variables that stand for a fixed, but unknown, value. They are typically distinguished by being capitalized, and should be treated as constants.
described in the Appendix), the identities should be obvious. In principle the relevant identities can be codified as part of the introduction of new notation. For example, the commutativity of addition and transitivity of relational on numbers could be described as

$$\forall(a, b : \text{Numeric}) \parallel a+b \equiv b+a$$

and

$$\forall(a, b, c : \text{Real}) \parallel a,b,c \in \text{Real} \parallel a<b<c \Rightarrow a<c$$

(We assume here that type \text{Numeric} includes the complex numbers, so that we restrict the transitivity property to the reals).

A predicate logic establishes identities and implications between predicates, while the predicate transformer relates predicates and statements. The predicate transformer $wp$ and a predicate logic together comprise a programming calculus equivalent to the corresponding programming logics.

### 2.3.1. Initial definitions of the predicate transformer $wp$

For each $\langle stmt S \rangle$ of our initial grammar $G$ we now sketch the definition of predicate transformer $wp.(S,R)$, for arbitrary $R$.

**Assignment.** $\langle idl \tilde{v} \rangle := \langle exp e \rangle$

$$wp.\langle \tilde{v} := e', R \rangle \equiv \text{dom.} \quad e' \land R^\tilde{v}_e$$  \hspace{1cm} (2.6)

The assignment of a structured value $e$ to a list of identifiers $\tilde{v}$ generalizes simple and multiple assignment statements but increases the complexity of the associated predicate transformer since a list of values $\tilde{e}$ has to be formed from the value $e$. The predicate $\text{dom.} \langle \text{exp e} \rangle$ is defined, analogous to $wp$, in a syntax-directed manner by the appropriate type definitions and is \textit{true} only in the states in which evaluation of $e$ is defined, and \textit{false} otherwise. For example, type \text{Numeric} might include this definition for division ($/$):

$$\text{dom.} \langle \text{exp e}_0 \rangle / \langle \text{exp e}_1 \rangle \equiv \text{dom.} \quad e_0' \land \text{Numeric}.e_0$$

$$\land \quad \text{dom.} \quad e_1' \land \text{Numeric}.e_1 \land e_1 \neq 0$$

For an appropriate $\langle \text{exp e} \rangle$ this definition and the predicate logic can be used to simplify $\text{dom.} e$. For example,

$$\text{dom.} \quad \langle a / S \rangle \equiv \text{dom.} \quad a' \land \text{Numeric}.a$$

$$\land \quad \text{dom.} \quad S' \land \text{Numeric}.S \land S \neq 0$$

$$\equiv \text{dom.} \quad a' \land \text{Numeric}.a$$

We have already stated our assumption that all variables denote a value, so we can always discharge $\text{dom.} \langle a' \rangle$. In addition we expect that the type inference system will guarantee that $\text{Numeric}.a$ holds throughout the scope of $a$. In practice, then, the domain predicate should not be as cumbersome as it appears.

The list of values $\tilde{e}$ is composed of the consecutive tuple members of $e$, matching the length of $\tilde{v}$. For example, the assignment $a, b := e$ yields

$$\tilde{v} = a, b$$

$$\tilde{e} = e.0, e.1$$
In the case that \( e \) is the tuple value \((a, b)\), the list \( \vec{e} \) can be simplified to \( \vec{e} = a, b \) (by properties of extractors on tuples).

The assignment is well-defined only if the length of the identifier list \( \vec{v} \) and the number of components in the tuple value agree; observe that \( a, b, c := (e_0, e_1) \) can never be correct. This condition can be ensured syntactically by a type system, so we can assume that for each \( v_i \in \{\vec{v}\} \) there is a corresponding \( e_i \) in \( \vec{e} \).

The multiple assignment statement described in [Gries78],

\[
v_0, \ldots, v_{n-1} := e_0, \ldots, e_{n-1}
\]

(2.7)
can be expressed here as

\[
v_0, \ldots, v_{n-1} := (e_0, \ldots, e_{n-1})
\]

(2.8)
which, by simplification, leads to the same substitution list (\( \vec{e} \) for \( \vec{v} \)) given for (2.7), although multiple occurrences of the same identifier on the left are not permitted in our case.

**Remark**

It is unfortunate that the two sides of (2.8) do not ‘‘appear’’ the same, as they do in (2.7); this is due to the fact that we have chosen different display forms for a tuple of identifiers (no parentheses) and a tuple of values (where parentheses are required because tuples may be nested).

The predicate \( R^\vec{y} \) in (2.6) denotes the simultaneous substitution of the \( e_i \) for the \( v_i \) in \( R \). We define this substitution by analysis of \( \langle \text{exp} \ R \rangle \) into components. For the simple case, if \( R = \langle id \ x \rangle \), then

\[
R^\vec{y} = x^\vec{y} = \begin{cases} x & \text{if } x \not\in \{\vec{v}\} \\ e_i & \text{if } x = v_i \end{cases}
\]

which is uniquely defined because the \( v_i \) are distinct (recall \( x \) is a label and denotes the textual representation of an identifier).

If \( R \) is not simple, then \( R \) has the form

\[
( \phi \langle \text{exp} \ E_0 \rangle \cdots \langle \text{exp} \ E_{n-1} \rangle )
\]

for some right hand side of production \( \langle \text{exp} \rangle \) named by \( \phi \). The substitution applies to each of the component expressions \( E_i \):

\[
R^\vec{y} = (\phi \ (E_0)_{\vec{y}} \cdots (E_{n-1})_{\vec{y}})
\]

Straight syntactic substitution might lead to substitution of a bound occurrence in \( R \) of an identifier \( v \in \vec{v} \), so we always restrict substitution of \( \vec{v} \) to occurrences of identifiers that are free in \( R \), i.e.

\[
R^\vec{y} = R^{\vec{y}'}
\]

where \( \vec{y}', \vec{e}' \) consist only of those \( v_i, e_i \) that satisfy \( v_i \in \text{fv.} R \).

Finally, there is the possibility of an identifier that occurs free in one of the \( e_i \) being ‘‘captured’’ by an enclosing scope with a bound identifier of the same name. Thus, if \( R \) is a scope binding identifiers \( \vec{x} \) some of which may occur free in \( \text{fv.} e \), then the \( \vec{x} \) must be consistently renamed throughout the scope to \( \vec{y} \), chosen not to conflict with \( \text{fv.} e \) or \( \vec{v} \). That is, if
\[ R = (\langle \text{idl } \bar{x} \rangle \parallel \langle \text{exp } P \rangle \parallel \langle \text{exp } W \rangle) \text{ then} \]
\[ R^\bar{y} = (\bar{x} \parallel P \parallel W)^\bar{y} = (\bar{y} \parallel P^\bar{y} \parallel W^\bar{y}) \]
provided \( \bar{y} \) has the same length as \( \bar{x} \) and satisfies
\[ ((\bar{y}) \cup f v . e) \cap (\bar{y}) = \emptyset. \]
This completes the inductive definition of substitution and the predicate transformer for assignment.

**Composition.** \( \langle \text{stmt } S_0 \rangle; \cdot \cdot \cdot ; \langle \text{stmt } S_{n-1} \rangle \)
\[ \text{wp} . (\langle S_0 \rangle; \cdot \cdot \cdot ; S_{n-1} \rangle, R) = \text{wp} . (\langle S_0 \rangle , \text{wp} . (\langle S_1 \rangle ; \cdot \cdot \cdot ; S_{n-1} \rangle, R)) \] (2.9)
The predicate transformer of statement composition is the functional composition of the statement predicate transformers. Statement composition is associative since function composition is.

**Choice.** choose \( \langle \text{idl } \bar{v} \rangle \) so \( \langle \text{exp } P \rangle \)
\[ \text{wp} . (\langle \text{choose } \bar{v} \text{ so } P \rangle , R) = \exists(\bar{v} \parallel \text{true} \parallel P) \land \forall(\bar{v} \parallel P \parallel R) \] (2.10)
There must be some choice for \( \bar{v} \) that satisfies \( P \), and for any such choice \( R \) must hold. The choice statement is a specification construct. It specifies a final state without indicating "how" the state is to be achieved, only which variables may change. The choice statement is a non-standard programming language construct since it is not generally implementable. We take a moment to examine its standing with regard to the programming logic.

Dijkstra [Dijkstra76] defines five "healthiness" conditions (P1-P5 below) that are expected of the predicate transformer \( \text{wp} \) in the logic. Let \( S = \text{choose } v \) so \( P \) for arbitrary \( v \) and predicate \( P \) in the following.

P1: \[ \text{wp} . (S, \text{false}) = \exists(v \parallel \text{true} \parallel P) \land \forall(v \parallel P \parallel \text{false}) \]
\[ = \{ \text{by case of } P \} \]
\[ (\text{false } \lor \text{true}) \lor (\text{true } \land \text{false}) \]
\[ = \text{false} \]
P1 excludes "miracles". It ensures that the choice statement, as a predicate transformer, does not "break" the programming logic by acting as a statement that can establish \( \text{false} \) from some existing initial state. Such a statement would be a miracle since its postcondition could then be used to establish any result.

Now suppose we have predicates \( Q \) and \( R \) such that \( Q \Rightarrow R \). Then choice is monotonic since

P2: \[ \text{wp} . (S, Q) = \exists(v \parallel \text{true} \parallel P) \land \forall(v \parallel P \parallel Q) \]
\[ \Rightarrow \exists(v \parallel \text{true} \parallel P) \land \forall(v \parallel P \parallel R) \]
\[ = \text{wp} . (S, R) \]
For any \( Q \) and \( R \), \( Q \Rightarrow Q \lor R \) and \( R \Rightarrow Q \lor R \). Using (P2) twice with consequent \( Q \lor R \) we always have the rule of disjunction:
P3: \( wp.(S, Q) \lor wp.(S, R) \Rightarrow wp.(S, Q \lor R) \lor wp.(S, Q \lor R) \)
\[ = \quad wp.(S, Q \lor R) \]

The rule of conjunction is also satisfied:

P4: \( wp.(S, Q \land R) = \exists (v \parallel true \parallel P) \land \forall (v \parallel P \parallel Q \land R) \)
\[ = \exists (v \parallel true \parallel P) \land \forall (v \parallel P \parallel Q) \land \forall (v \parallel P \parallel R) \]
\[ = \quad wp.(S, Q) \land wp.(S, R) \]

The fifth healthiness condition, P5, requires that \( wp \) be continuous over families of monotonic predicates. For any family of predicates \( Q_i \) satisfying \( \forall (i \parallel i \geq 0 \parallel Q_i \Rightarrow Q_{i+1}) \) P5 requires

\[ \exists (i \parallel 0 \leq i \parallel wp.(S, Q_i)) = wp.(S, \exists (i \parallel 0 \leq i \parallel Q_i)) \quad (2.11) \]

The choice statement fails to be continuous: for any \( P \) satisfied by an unbounded set of choices for \( v \), there is some family of predicates \( Q_i \) for which (2.11) does not hold. For example, the statement choose \( v \) so \( 0 \leq v \) can take \( v \) to be any non-negative integer. The family of predicates \( Q_i \triangleq 0 \leq v \leq i \) satisfies \( Q_i \Rightarrow Q_{i+1} \) but

\[ wp.(\text{choose } v \text{ so } 0 \leq v, \exists (i \parallel 0 \leq i \parallel 0 \leq v \leq i)) \]
\[ = \exists (v \parallel true \parallel 0 \leq v) \land \forall (v \parallel 0 \leq v \parallel \exists (i \parallel 0 \leq i \parallel 0 \leq v \leq i)) \]
\[ = \quad true \]
\[ \neq false \]
\[ = \exists (i \parallel 0 \leq i \parallel true \land false) \]
\[ = \exists (i \parallel 0 \leq i \parallel \exists (v \parallel true \parallel 0 \leq v) \land \forall (v \parallel 0 \leq v \parallel 0 \leq v \leq i)) \]
\[ = \exists (i \parallel 0 \leq i \parallel wp.(\text{choose } v \text{ so } 0 \leq v', 0 \leq v \leq i)) \]

The lack of continuity has the consequence that the predicate transformer for the repetitive statement (to be defined later) can not always yield the weakest precondition. In practice this should not be a problem since we rarely use the weakest precondition of the repetitive construct, but rather rely on a stronger invariant predicate of our own manufacture that is easier to use. Furthermore, for any finitary \( P \) (one that is satisfied by a finite number of values for \( \overline{v} \)), the statement is continuous.

Although there can be no general terminating implementation of the choice statement for arbitrary \( R \), whenever \( P \) can be implemented, a choice satisfying \( P \) can eventually be found by exhaustive (dovetail or parallel) testing of possible values. This approach is, of course, completely unsatisfactory in a practical program, so we are interested in restricted instances of the choose statement which can be implemented efficiently.

Consider, for example, the statement choose \( x \) so \( x = e \), for a simple \( \langle id x \rangle \) and \( \langle \exp e \rangle \). This is a particular form of \( P \) for which the predicate transformer (2.10) specializes to

\[ wp.(\text{choose } x \text{ so } x = e', R) \]
\[ = \exists (x \parallel true \parallel x = e) \land \forall (x \parallel x = e \parallel R) \]
\[ = \quad \text{dom}(e') \land \forall (x \parallel x = e \parallel R) \]
\[ = \quad \text{dom}(e') \land R^* \]

Since this is the predicate transformer for the assignment \( x := e \), in a sense to be made precise later, the assignment implements the restricted form of choice.

A different kind of specification statement and its predicate transformer are presented in [Morgan86]. The statement has the form \( \overline{w} : \langle pre, P \rangle \) where \( \overline{w} \) is a list of identifiers, and \( pre \) and
$P$ are pre- and post-conditions, respectively. The interpretation is: assuming pre holds, choose values for identifiers $\bar{w}$ so that $P$ holds. The predicates used in [Morgan86] relate initial to final states following [Jones80], rather than denoting a single set of states as is the case for our predicates. In our setting, Morgan's predicate transformer simplifies to

$$wp.(\bar{w} : (pre, P)', R) = pre \land \forall(\bar{w} \parallel P \parallel R)$$

which looks similar to our definition (2.10) but has a subtle difference. The specification statement we defined has no clause analogous to pre but instead requires $P$ to be satisfiable. It is precisely in states where $pre \neq false$ and $P = false$ that Morgan's statement violates the law of the excluded miracle (P1). Consequently, a program $S$ developed with Morgan's specification statement must be checked to insure that it does not depend on miracles, and this is the case precisely when $wp.(S, false) = false$. The additional obligation is avoided in our case because the choice statement does not violate P1.

\[ Block. \quad (\langle idl \bar{v} \rangle \parallel \langle exp P \rangle \parallel \langle stmt S \rangle) \]

\[
wp.(\langle \bar{v} \parallel P \parallel S' \rangle, R) = \exists(\bar{v} \parallel true \parallel P) \land \forall(\bar{v} \parallel true \parallel wp.(S, R)_{\bar{v}})_{\bar{v}} \quad (2.12)
\]

The block statement is essentially a scope that surrounds the introduction of the bound variables $\bar{v}$ satisfying the domain predicate $P$ and the following statement $S$. Consequently the predicate transformer is closely related to the predicate transformer for the choice statement and differs only in that it binds all occurrences of $\bar{v}$ in $S$. The predicate transformer for a block introduces a list of fresh identifiers $\bar{y}$ corresponding in length to $\bar{v}$ but distinct from $\bar{v}$, $fv.R$ and $fv.S$. The inner substitution of $\bar{v}$ by $\bar{y}$ serves to "hide" free occurrences of $\bar{v}$ in $R$ from capture by the scope. The universal quantification can only be satisfied when the weakest precondition of $S$ with respect to $R_{\bar{y}}$ is independent of $\bar{v}$. The outer substitution restores the free occurrences of identifiers to their original names. The substitutions need not be performed on the existential clause, since $R$ does not occur in it.

The definition for the general case of a block can be specialized to instances for which it is simpler. If the identifiers in $R$ are distinct from the $\bar{v}$, that is $fv.R \cap \bar{v} = \emptyset$ then (2.12) simplifies to

\[
wp.(\langle \bar{v} \parallel P \parallel S' \rangle, R) = \exists(\bar{v} \parallel true \parallel P) \land \forall(\bar{v} \parallel true \parallel wp.(S, R)) \quad (2.13)
\]

and if, in addition, we have $P = true$ we obtain

\[
wp.(\langle \bar{v} \parallel true \parallel S' \parallel true \rangle, R) = \forall(\bar{v} \parallel true \parallel wp.(S, R)) \quad (2.14)
\]

This last simplification would correspond most closely to a predicate transformer definition of a standard scope without initialization of the variables introduced. Recall that from our point of view all variables are always valued so that the closest we can come to an uninitialized variable is a variable with an unconstrained value. The term $\forall(\bar{v} \parallel true \parallel wp.(S, R))$ can be read as "$wp.(S, R)$ is independent of $\bar{v}$". Note that any block can be placed into a form for which (2.14) applies by judicious relabeling of the $\bar{v}$ and application of the identity (2.1) to replace a stronger predicate $P$ by $true$. 
Value-yielding block. \((\langle idl \bar{x} \rangle \parallel \langle exp \ P \rangle \parallel \langle stmt \ S \rangle \ yield \langle exp \ E \rangle)\)

A valblock is, of course, an expression. But since the expression involves an internal state and a statement, we outline its interaction with the programming logic.

If \(F\) is a valblock expression of the form above, we take a specification \(\langle exp \ f \rangle\) for \(F\) to be another expression that does not include a valblock. We use a predicate transformer definition to determine \(\text{dom.} \ F\), the largest domain on which \(F\) agrees with \(f\). This lets us use \(f\), which is simpler to manipulate and understand since it involves no valblocks, in predicates but allows \(F\) to take its place in program expressions whenever \(\text{dom.} \ F\) holds. This is the sense in which a valblock is a refinement construct. For example, the statement \(x := f\) is an easy way to achieve the specification \((R, R \land x = f)\). If we have \(R \Rightarrow \text{dom.} \ F\), then we may use the statement \(x := F\) to achieve the same specification.

For \(F\) and \(f\) as above we define the domain of \(F\) as

\[
\text{dom.} \ F = \exists(x \parallel true \parallel P) \land \forall(x \parallel P \parallel wp.(S, f=E))
\]

For instance, if we take the specification \(f\) to be \(a \ast b\), and we take the expression \(F\) to be \((x \parallel 0 \leq x \leq a \parallel x := x + b \ yield \ x)\), then

\[
\text{dom.} \ F = \exists(x \parallel true \parallel 0 \leq x \leq a)
\land \forall(x \parallel 0 \leq x \leq a \parallel wp.(\langle x := x + b \rangle, a \ast b = x))
\]

\[
= 0 \leq a \land \forall(x \parallel 0 \leq x \leq a \parallel a \ast b = x + b)
\]

\[
= 0 \leq a \land (a = 0 \land b = 0)
\]

\[
= a = 0 \land b = 0
\]

Only when \(a\) and \(b\) are both zero does \(F\) behave like the expression \(f\).

We have mentioned that for any \(\langle exp \ E\rangle\) we require referential transparency of \(E\). That is, if \(\bar{v} = \text{fv.} \ E\), then for any two choices of values for all identifiers on which \(\bar{v}\) are the same, \(E\) must yield the same result. The syntactic constraint (2.5) given in section 2.2.6 insures that \(F\) has no side-effects, but the presence of nondeterministic statements in our notation means that a valblock \(F\) might not be uniquely defined as a function of \(\text{fv.} \ F\). The "domain" definition we have given for \(F\) guarantees that this is not the case whenever \(\text{pre.} \ F\) holds; then the expression \(F\) is uniquely defined since specification \(f\) is.

2.3.2. Some properties of scopes and statements

We list some properties of a programming notation that hold for all the initial definitions introduced thus far and must be met by any new notation defined.

The bound occurrences of variables in an expression scope may be systematically relabeled without affecting the meaning of the scope, provided the relabeling doesn't capture free occurrences of identifiers in the scope body. That is, for any

\(\langle \text{scope} \langle idl \bar{x} \rangle \langle exp \ P \rangle \langle exp \ W' \rangle \rangle\)

and \(\langle idl \bar{y} \rangle\) of the same length as \(\bar{x}\) and satisfying \(\bar{y} \cap \text{fv.} \ W = \emptyset\), we have

\[
(\bar{x} \parallel P \parallel W) = (\bar{y} \parallel P'_x \parallel W'_x)
\]

(2.15)

The definition of substitution in predicates (expressions) that follows (2.6) can easily be extended over a \(\langle stmt \ S \rangle\), by distributing the substitution over all the subproductions of \(S\). Since \(S\) is composed only of scopes, expressions, identifier and statement productions, and
substitution is defined for all of these, this completes the definition. Note that, in general, we can only substitute identifiers for identifiers in statements because they may contain \langle id \rangle subproductions that are not part of an \langle exp \rangle.

We can now extend \( f v \) over a Hoare triple \( \Xi \):

\[
fv \cdot (\langle \exp P \rangle \ \langle \stmt S \rangle \ \langle \exp Q \rangle) = fv \cdot P \cup fv \cdot S \cup fv \cdot Q
\]

and require that for any \( \langle idl \ \bar{X} \rangle \) and \( \langle idl \ \bar{Y} \rangle \) of the same length with \( \bar{Y} \cap fv \cdot \Xi = \emptyset \), we have that \( \Xi \) is a valid Hoare triple precisely when \( \Xi_{\bar{Y}} \) is. That is,

\[
(P \ \{ S \ \} \ Q) \equiv (P_{\bar{Y}} \ \{ S_{\bar{Y}} \} \ Q_{\bar{Y}})
\]

In addition, whenever we have a \( \langle stmt S \rangle \) and a predicate \( R \) such that \( def \cdot S \cap fv \cdot R = \emptyset \), we require that

\[
\{ R \wedge wp \cdot (S, true) \} \ \{ S \ \} \ \{ R \}
\]

be a theorem of the programming logic. In words, if a statement updates a set of variables disjoint from the identifiers occurring in the predicate \( R \), then \( R \) is invariant over the statement in any state where the statement terminates.

### 2.4. Semantics

All the rules to be used in the derivation of programs are given as a formal system. The programming logic connects pairs of predicates and programs while the predicate calculus transforms expressions by identities and rules of inference.

Since the ultimate goal of program derivation is to arrive at a particular proof of the problem that is executable by machine, what confidence do we have that the formal system properly represents computation? It is not the case that every program text we derive that meets our specification is executable, so there will be no direct reflection of our statements and expressions as computations. We have an idealized model of computation in mind, however, when we derive a program, and certain restrictions of that model will correspond to machine computations. In this section we outline this model, and then discuss the correspondence between our formal system and the model.

#### 2.4.1. A relational model for programs

Our expression language is a heterogeneous word algebra that contains every \( \langle exp \rangle \), as given by the type definitions, partitioned into equal terms by the identities set forth in the the type definitions. There are many models for such an algebra, but from the point of view of manipulation and simplification of predicates, we can only use the defining identities. We do not assume values are different if we can’t prove them identical, so we adopt initial models. These models associate a value with each class of identical terms of the word algebra, but keep unrelated (by identities) classes associated with different values. The set of values in the model is called the carrier set. Typically our type definitions define “natural” data types for which there is some standard model, such as the standard model of arithmetic for the type Numeric, for which the carrier is the complex numbers Complex.

For now it might be simplest to think only of type definitions Integer and Boolean, which define, say, integer arithmetic, and predicate logic. The carrier sets are the integers \( \mathbb{Z} \) and the boolean values \( \{ true, false \} \). We add one extra value, \( \mathbb{Z} \), to the model that will stand for (the
value of) an undefined expression, so that we can treat all expressions as total. For example, we will take \(5/0\) to have value \(\bot\). We make the following assumptions about the value \(\bot\):

(a) All expressions except Boolean are strict with respect to \(\bot\). Thus, a tuple containing value \(\bot\) has value \(\bot\), and for any non-propositional function \(f, f(\bot) = \bot\).

(b) The propositional connectives are non-strict, but "conservative", in their extension on \(\bot\). So \(false \land \bot = false\) but \(true \land \bot = \bot\) and \(\bot \land \bot = \bot\). See [Jacobs85] for a precise definition of the propositional connectives under this three-valued interpretation.

Any expression in which no identifiers occur free is a ground term and denotes a unique value in this model. The reduction from a ground term to its associated value is called evaluation. To evaluate expressions in which there are free occurrences of identifiers, we need to associate identifiers with values.

**Definition (State)**

A state \(s\) is a function from identifiers to values. All states have the same domain, the set of all identifiers, but each may have a unique range (assignment of values). The values are drawn from the carrier sets modeling the expression language, with the exception of the value \(\bot\), which may never be associated with an identifier. We say an identifier in a state is valued, since it can never be associated with \(\bot\). The value of identifier \(x\) in state \(s\) is \(s.x\), and the state \((s; x : a)\) is a state that differs from \(s\) only on the value of \(x\), which is \(a\). So \((s; x : a).x = a\) and \((s; x : a).y = s.y\), for \(y \neq x\). Here, as in the following, \(s, s', \ldots\) are states, \(x, y\) are identifiers, and \(a\) is a value.

**Definition (Set of all states)**

Let \(S^\bot\) be the set of all possible assignments of values to all possible identifiers, along with a distinguished extra state \(\bot\). The extra state represents non-termination; it is a state that can never be achieved by a program. The difference between \(\bot\) and \(\bot\) is that \(\bot\) models a value that can never be associated with an identifier, while \(\bot\) models a state that can never be achieved by a program.

**Definition (Evaluation of an expression)**

Let \(\bar{v}\) be the list of all identifiers in a state \(s \in S^\bot\), and let \(\bar{a}\) be their corresponding values (so \(a_i = s.v_i\)). The extension of the state function \(s\) over expressions \(\langle expE \rangle\) is written \(s.E\) and denotes evaluation of the ground term \(E^\bar{v}_a\). If the value of \(E^\bar{v}_a\) is undefined, then \(s.E = \bot\). Evaluation of predicates, as we indicated, may not be strict, but for all expressions \(E\), including predicates, \(\bot.E = \bot\). A predicate \(P\) is satisfied by a state \(s\) if and only if \(s.P = true\).

The essential effect of a statement is the alteration of state. If statements were deterministic, we could model them as functions from states to states \((S^\bot \rightarrow S^\bot\)). Since nondeterministic statements (like choice) can have many possible final states for each initial state, we might model statements as functions from states to sets of states \((S^\bot \rightarrow \mathcal{P}(S^\bot))\). But this makes the domain and range different and makes composition more difficult. Hence we take the approach of [Hoare74] and adopt binary relations as our model of statements. The extra element \(\bot\) was introduced by Plotkin [Plotkin76] to model non-termination, but it also has a use very similar to the value \(\bot\): to totalize partial relations. Each statement, and hence each program corresponds
to some program relation.

**Definition (Program Relations)** (c.f. [Hoare74, Hoare85])

We view a relation $R$ as a set of pairs. If $(s, s') \in R$, then $R$ relates $s$ to $s'$, and $s$ is in the domain of $R$. Not every relation is a program relation; $R$ defines a program relation if

(a) $R$ is a subset of $S^L \times S^L$. For each $(s, s') \in R$, initial state $s$ may lead to final state $s'$.

(b) The state $\bot$ is only and exactly related to $\bot$: $\forall (s' \parallel (\bot, s')) \in R \parallel s' = \bot$. This reflects the fact that non-termination can never be "undone". But note that tuples of the form $(s, \bot)$ are not precluded, and model non-termination from initial state $s$.

(c) $R$ is total, i.e. $S^L = (s, s' \parallel (s, s') \in R \parallel s)$. If $R$ is a partial relation, then the totalized relation $R^+$ relates any $s \in S^L$ that is not $\bot$ or in the domain of $R$ to any final state $s' \in S^L$ whatsoever (including $\bot$). More precisely,

$$R^+ = R \cup (\bot, \bot) \cup (s, s' \parallel \neg \exists (s'' \parallel (s, s'') \in R) \land s \neq \bot \parallel (s, s'))$$

$R^+$ corresponds to completely unconstrained behavior (including nontermination) outside the domain of $R$. We want program relations to be total because it makes possible behavior more explicit and makes composition of relations unambiguous.

We are now ready to give a model-theoretic formulation of the programming language statements in $G$. For each $\langle \text{stmt } S \rangle$, we define a corresponding program relation $R_S$. We start with the two basic statements, assignment and choice. For $\langle \text{id } x \rangle, \langle \exp E \rangle$ and predicate $P$ we define the corresponding relations (the extension to fully general assignment and choice is straightforward).

$$R_{\text{:= } E} \triangleq (s \parallel s.\text{dom}(E') \parallel (s, (s; x : (s.E))))^+$$

$$R_{\text{choose } x \text{ so } P} \triangleq (s, a \parallel (s; x : a), P \parallel (s, (s; x : a)))^+$$

Under the assumption that $s.\text{dom}(E') \equiv (s.E \neq \bot)$, i.e. that dom$(E')$ correctly characterizes the domain of $E$, neither of these relations can contain a state with a value $\bot$, since $s$ and $a$ vary over defined states and values and dom$(E')$ guards the inclusion of value $s.E$. Thus the relations are subsets of $S^L \times S^L$. Since they are totalized, they are program relations.

Other statements in $G$ are constructed from simpler parts by statement composition, or encapsulation of a variable in a block. Given relations $R_S$ and $R_T$ corresponding to $\langle \text{stmt } S \rangle$ and $\langle \text{stmt } T \rangle$, we model $S; T$ as

$$R_{S; T} \triangleq (s, s' \parallel \exists (t \parallel \text{true} \parallel (s, t) \in R_S \land (t, s') \in R_T) \parallel (s, s'))$$

If $S$ can reach $t$ from $s$ and $T$ can reach $s'$ from $t$, then $S; T$ can reach $s'$ from $s$. $R_{S; T}$ defines a program relation if $R_S$ and $R_T$ are program relations, since (a) $s, s'$ are both drawn from $S^L$, and (b) $(\bot, \bot) \in R_S$, $(\bot, \bot) \in R_T$ so $(\bot, \bot) \in R_{S; T}$ and there are no other tuples $(\bot, s')$ in $R_S$ or $R_T$ for any $s' \neq \bot$, and (c) $R_S$ is total hence $R_{S; T}$ is total.

A block binds a variable $x$ over a statement $S$. The corresponding relation $R_{(x \parallel \text{true} \parallel S)}$ leaves the value of $x$ unchanged, but behaves like $R_S$ with any possible initial value $a$ for $x$.

$$R_{(x \parallel \text{true} \parallel S)} \triangleq (s, s', a \parallel ((s; x : a), s') \in R_S \parallel (s, (s'; x : (s.x))))$$
$R_{(x \mapsto \text{true} \mid S)}$ is a program relation if $R_S$ defines a program relation. All states in the relation are valued, since $a$ is always valued, and $s \cdot x$ is valued whenever $s \neq \perp$, in which case $s'$ is also $\perp$. Furthermore, $R_{(x \mapsto \text{true} \mid S)}$ is total since $R_S$ is total so $(s; x : a)$ covers all states in $S^{\perp}$.

By induction over (stmt) we conclude that $R_S$ defines a program relation for any statement $S$. We have only pointed out correspondences between the notation and the model for the statements introduced thus far. With every type definition there are new values to incorporate in the model and new statements to model with program relations.

2.4.2. Consistency of the programming logic and relational model

A predicate $P$ represents the set of states $(s \p P \cdot s \p s)$. Let a program $S$ be represented by the relation $R_S$ constructed from the corresponding definitions. If $(P \cdot S \cdot Q)$ is a theorem of the programming logic then the soundness of the logic relative to the model rests on a demonstration that $R_S$ relates any state that satisfies $P$ to a set of states that each satisfy $Q$. That is

$$\forall(s \p P \cdot s \p s \cdot P \cdot s \p s \p s \p (s, s') \in R_S \p Q \cdot s')$$

(2.16)

We should choose $P$ in (2.16) to be $wp \cdot (S, Q)$ so that any $P_0$ such that $P_0 \Rightarrow wp \cdot (s, Q)$ will also satisfy (2.16), since for all $s, s, P_0 \Rightarrow s \cdot (wp \cdot (S, Q))$. To demonstrate that $wp$ really corresponds to the weakest precondition, we also have to examine the relation in the reverse direction. Each initial state for which all final states are in $Q$ should be in $wp \cdot (S, Q)$:

$$\forall(s \p (s' \p (s, s') \in R_S \p s '' \cdot Q) \p s \cdot (wp \cdot (S, Q)))$$

A proof of completeness of the logic consists of a demonstration that if a set of states $P$ is related to a set of states $Q$ by a program relation $R_S$ as constructed above, then there are predicates that represent $P$ and $Q$ and $(P \cdot S \cdot Q)$ is a theorem of the programming logic. This is more difficult to prove, and, depending on the axiomatization of the predicate logic, sometimes not even possible. Even with an expression language as simple as arithmetic the completeness of any finite axiomatization is precluded, so that "relative completeness" is the more usual notion in programming logics. A logic $A$ is relatively complete with respect to an included logic $B$ if completeness can be proven under the assumption that $B$ is complete, i.e. that any true formula of $B$ can be proven.

We will not undertake such proofs here, even for the small formal system we have developed thus far. When we extend the programming notation we will have to extend the formal system to describe the properties of the new notation, and the soundness and (relative) completeness of the new system will come into question. In the general case the proofs above will have to be completely redone for the extended formal system. A topic that is not investigated here is under what conditions the new notation leads to an extension of the formal system that is conservative, meaning that the soundness and completeness of the extended system can be simply demonstrated from the soundness and relative completeness of the base system and the extension.

2.4.3. Modeling specifications

The relational model can also be used to represent specifications. A specification $(P, Q)$ requires that any initial state satisfying $P$ is related only to states satisfying $Q$. The most unconstrained such relation simply relates each state that satisfies $P$ to every state that satisfies $Q$:
\( R_{(p,q)} = \{ s, s'' \parallel (s.p) \land (s'.q) \parallel (s, s') \} \)

Outside of \( P \) the specification does not prescribe behavior, so we may choose any behavior that suits us. The most unconstrained relation is obtained from the totalization \( R_{(p,q)} \) [Hoare85], which is also a program relation.

Compared to a specification relation, the relations corresponding to the statements of our programming notation are rather small "pre-packaged" relations. We could view our job to be the arrangement of these building-block relations to arrive at a final relation \( R_S \) that is a subset of the relation \( R_{(p,q)} \). This is because \( R_S \subseteq R_{(p,q)} \) means that every \( (s, s') \in R_S \) is also in \( R_{(p,q)} \) and hence meets the specification. For program relations \( R_S \) and \( R_T \), \( R_S \subseteq R_T \) means \( S \) is more deterministic than \( T \).

The relational calculus is a very elegant tool that can help build \( R_S \) from \( R_{(p,q)} \), in the same way that \( wp \) can help derive \( S \) from \( P \) and \( Q \). This is precisely what Hoare and He have explored in [Hoare85]. The only drawback of this approach is that the definitions of the various program relations make for rather more work in the calculation and simplification steps than predicate transformer approach. For example, consider the verification that \( x := x+1; y := y-1 \) satisfies the specification \( (x+y = 0, x+y = 0) \). With the predicate transformer formulation we have to demonstrate that

\[
 x+y=0 \Rightarrow \text{dom.}(x+1) \land (\text{dom.}(y-1) \land (x+y=0)_{y-1})_{x+1}
\]

while the relational definition requires us to simplify and demonstrate

\[
 R_S = \{ s, s'' \parallel (s.x+s.y = 0) \land (s',x+s',y=0) \parallel (s, s') \}^+ 
\]

with

\[
 R_S = \{ t \parallel \text{true} \} \quad \{ (s, t) \in \{ s \parallel s.\text{dom.}(x+1) \parallel (s, (s; (s.x+1))) \} \}
\]

\[
 \land \{ t, s' \parallel \{ s \parallel s.\text{dom.}(y-1) \parallel (s, (s; (s.y-(s.y-1))) \} \}
\]

\[
 \parallel (s, s') \}
\]

Although many of our ideas are easier to express with the relational calculus, the practice of programming still seems simpler to us using predicate transformers, predicate logic and predicates that denote states. This is why we have chosen, for now at least, to continue with this approach. Methodologies based on predicates that describe relations are investigated in [Hehner86, Jones80, Hoare85].

### 2.4.4. Modeling implementations

A final point we can make about the relational model is that it can also capture implementations in the form of partial relations. By an implementation \( M \) of a program construct \( S \) we mean the approximation to the "ideal" program construct \( S \) offered on a given machine or in a given programming language. The implementation reflects the finite capabilities of a machine, or the limitations of a representational shortcut. But on some domain \( P \), the implementation does agree with the ideal program relation \( R_S \). We define \( P \downarrow R \) to mean the restriction of relation \( R \) to domain \( P \):

\[
 P \downarrow R = \{ s, s'' \parallel (s, s') \in R \land s.\text{P} \parallel (s, s') \}
\]

which yields a partial relation. We say the implementation \( M \) of \( S \) with applicability domain \( P \) is modeled by relation \( P \downarrow R_S \). If \( S \) is a solution to the specification \( (Q, R) \), then \( M \) is adequate if \( Q \Rightarrow P \) and can be used in place of \( S \) and
\[ P \downarrow R_S \subseteq R_S \subseteq R_{(Q,R)} \]

Since many partial relations are contained in a program relation, the \textit{adequacy} requirement is very important. Otherwise the empty relation would implement any specification. We can profitably use implementations when our building block relations are idealized and very expressive, but are "underutilized" by individual programs. There can be many implementations \( M \) for a program \( S \), but only some will be adequate for a given specification that \( S \) satisfies.
3 A Type System

3.1. Types of Values in a Programming Notation

An effective programming notation provides operations on a variety of different values, each appropriate to a different aspect of problem description. A type definition organizes the introduction of new values and operations into the programming notation. Some programming languages make do with a fixed set of definitions (e.g. Pascal) although they may be liberally parameterized to yield a great variety of instances, while others (e.g. Clu, Ada, Modula) support the inclusion of new definitions as part of the language, typically in the form of an "abstract data type".

We feel that a type definition facility should be available to tailor the notation to a problem area, but not to individual problems. Since new notation and new properties are hard to learn and difficult to formulate consistently, we want to avoid their introduction in the process of program derivation. Most abstract data type definitions have more to do with identifying a set of operations for which there exists a convenient class of implementations than with a comprehensive introduction of new notation. By selecting a small but expressive set of type definitions we hope to provide fixed access and meaning to the values in the language that does not change during the development of programs, while specialized implementations developed for restricted patterns of use of the notation cater to program-specific efficiency considerations.

To fully integrate new notation requires that it be afforded the same status as all other notation in the language. This requirement, and the discussion of the previous paragraph leads us to an extra-lingual type definition facility so that the type definitions themselves are not part of the programming notation. We expect to use such a facility relatively infrequently to "generate" an appropriate programming notation.

Whether or not a type definition is extra-lingual, it typically serves the same functions:

1. It provides syntactic and semantic information to the programmer. At the least the programmer can learn the syntax of new operations from a type definition. The programmer may also be able to determine the usage of the notation from properties given in the definition. Usually this is in the form of comments, but in some cases equations may be given.

2. It provides implementation information to a compiler. The type definitions are arranged along the lines of the underlying implementation, and each definition is typically associated with a single default implementation.

3. It provides information to a type checking or a type inference system that syntactically identifies certain basic errors in a proof of correctness or violations of assumptions under which operations are implemented.

We discuss each of these functions in our approach.
Introduction of syntax and semantics. To extend the syntax of our programming language we add additional right-hand sides to the productions of grammar $G$. Functions extend $\langle exp \rangle$; statements extend $\langle stmt \rangle$. As before, these extensions must be accompanied by concrete syntax that defines the display form. The syntactic function $fv$ can be uniformly extended over all new function definitions with the rules in sec. 2.2.6. New statements are more varied in their syntactic components and may require individual extension of the functions use and def.

The new terms added to $G$ must also be incorporated into the programming calculus to make them usable in program derivations. For expressions this means we must provide a set of identities that may be used to simplify or rearrange the new terms. For a new statement $S$ we must define the predicate transformer $wp(S, R)$ for arbitrary predicate $R$ or provide a theorem about its definition (such as the loop invariant theorem).

Although a type definition will often introduce a function whose domain or range lies in another type, definitions are essentially independent and are organized around the (carrier) set of values characterized by the identities. These identities are used in program derivation and apply only to values of the appropriate type. For example, the identity $p \land q \equiv q \land p$ given in type $\text{Bool}$ holds only when $p$ and $q$ are boolean. Therefore the identity is properly written as the tautology

$$\forall(p, q : \text{Bool} \parallel true \parallel p \land q \equiv q \land p) \tag{3.1}$$

The type name $\text{Bool}$ is being used here as a predicate that characterizes the values of type $\text{Bool}$. Hence (3.1) corresponds to

$$\forall(p, q : \text{Bool}, p \land \text{Bool}, q \parallel p \land q \equiv q \land p)$$

Thus, in contrast to a "universal" theory (e.g. [Demers84]) that gives meaning to any operation applied to any value, the identities and definitions are all restricted to values of the appropriate domains. Consequently, to simplify an $\langle exp \rangle E$ to $E'$ by identity $E = E'$ in type $T$, it will be necessary (but not always sufficient) to establish that $E$ is a value of type $T$. With a small amount of extra information in a type definition, it will be possible to eliminate such steps from program proofs; a type inference system will ensure syntactically that all such predicates hold.

Implementation. Our basic approach has been to uncouple the type definitions from their implementations. This gives us greater freedom in defining mathematical data types that can, at best, be only partially implemented. It also allows us to use a smaller set of type definitions that are free from implementation artifacts—distinctions between values that are subject to identical properties (behave identically), but are segregated for the purpose of efficient implementation. Typical examples are long- and short-word integers, or, more controversially, integers $\text{Int}$, reals $\text{Real}$ and the complex numbers $\text{Complex}$. The operations and properties of all these values are the same from the point of view of program development, but their implementations are quite different in efficiency and accuracy. Implementation directives will be factored in later to address these concerns.

Type checking and inference. By restricting the form of type definitions we can ensure that it is a syntactic matter to determine whether operations are applied only to values of the right type. This is a matter of course in many programming notations, but may be accomplished with
varying degrees of success and in different ways. The goal is to determine for each operation the types of its arguments, ensuring that they lie within (an approximation of) the domain of applicability of the operation. A type inference system makes the determination based on the operations used and their known or constructed result types, solving for the unknowns. Depending on the type definitions, this may not be possible. A type checking system relies on declarations of the unknowns to verify that the argument types are correct. Most systems are a combination of the two. ML uses type inference almost entirely, while Pascal and ADA rely mostly on declarations. Some languages postpone the issue until run-time; values carry their own type and each operation is checked for type-correctness as it is encountered. Typically these languages do not require type declarations.

In mathematical notations there are few declarations, and a great deal of inference is left to the reader, who has the benefit of knowing what various formulas are intended to express. Mechanical type inference quickly becomes too hard or undecidable in the presence of the sophisticated type systems that accurately prescribe the types of values in mathematical notations.

For a given notation, there is a spectrum of type systems differing in the precision with which the domains of operations in the notation are described. At one end are coarse descriptions of the domains with simple rules relating the type of the domain to the type of the range. For example, integer division might be characterized as an operation that takes two values of type \textit{Int} and yields another value of type \textit{Int}. But a more precise description might categorized values as \textit{Zero} or \textit{Nonzero}. More accurate rules can now be given for integer division: a \textit{Zero} and \textit{Nonzero} value produce a \textit{Zero} value, a \textit{Nonzero} and a \textit{Nonzero} value yield a \textit{Nonzero} value, and these are the only rules for integer division (the other combinations are erroneous). However, developing these rules over other arithmetic operations will be problematic.

A standard approach to this problem is to require more declarations and use more sophisticated type rules, often ones that allow one to "compute" the result type from the argument types; this is the approach taken with the \textit{Pebble} type system [Burstall84], for example. As the domains more precisely characterize the operations, the type system characterizes more of the behavior of a notation, and, in the limit, type correctness becomes equivalent to program correctness. The best example of this approach is the \textit{Nuprl} system [Constable86] which is actually not a system for programming as much as a system for developing proofs. The extremely rich type system of \textit{Nuprl} allows true propositions to be viewed as types so that the rules of type correctness can be used to prove propositions. Hence the type system subsumes the need for a separate programming logic.

We will rely on a separate formal system to develop program correctness proofs, and our requirements of a type system are only that it simplify that process by automating steps that can be deduced directly from the syntax of the program. Our interests, then, are in type systems for as large a class of notations as possible, for which type inference is always possible. Since many mathematical notations are polymorphic —having operations defined over more than one type— a good candidate is the relatively simple polymorphic type system presented in [Milner78]. For this system there exists a simple inference algorithm [Milner78] that can be adapted to operate in a setting where new notation can be introduced. Although many of the reasons for a more complex type system disappear when implementation information is not part of a type, in practice a more sophisticated type system (perhaps along the lines of HOPE
language [Burstall80]) might nevertheless be preferable. Or we might have to compromise our objectives and include certain type declarations with variables.

We have mentioned that type correctness can be seen as an approximation of program correctness. The sense in which we rely on this is as follows. If the inference system is able to find a typing for the free variables in a program segment, then a program proof of that segment need never mention any type predicates — the type correctness proof is factored out, of the program correctness proof by the type inference system. In the contrapositive sense, if no such typing can be inferred then the program fragment can not be correct.

3.1.1. Type definitions

We turn now to the mechanism for including the typing rules with the presentation of new notation. Each type definition will introduce a sort: a name for the type of values being defined. Within a definition the \( \langle \text{exp} \rangle \) and \( \langle \text{smt} \rangle \) nonterminals of grammar \( G \) are extended with new abstract syntax following the approach of section 2.2. Each nonterminal in a new production will be attributed with a sort that characterizes the type of its values. The attached sort names define a signature for the new \( \langle \text{exp} \rangle \) and \( \langle \text{smt} \rangle \) productions.

To illustrate the attachment of sorts, suppose we are defining type \( \text{Bool} \) (the name \( \text{Bool} \) is the sort). The abstract and concrete syntax to define the logical \emph{and} instruction is

\[
\langle \text{exp} \rangle ::= \quad \langle \text{and} \ \langle \text{exp} \ p \rangle : \text{Bool} \ \langle \text{exp} \ q \rangle : \text{Bool} \rangle : \text{Bool}
\]

\[
\sim p \land q
\]

Each nonterminal is followed by a colon and a sort. The concrete syntax is separated from the abstract syntax by \( \sim \) and uses the labels introduced in the abstract syntax to indicate the display form of the function. The three terms in the definition are all labeled with sort \( \text{Bool} \).

A more complicated example involves scope. The only component of a scope that is allowed to vary in type of value is the final component \( W \), the range. If \( W \) is of sort \( \sigma \) then the complete scope is of sort \( \text{Scope}(\sigma) \). Recall, however, that a scope, on its own, has no value, so this definition is for the convenience of attaching a signature to the components of functions like existential or universal quantification. For example, the existential quantifier can be introduced as

\[
\langle \text{exp} \rangle ::= \quad \langle \text{exists} \ \langle \text{scope} \ X \rangle : \text{Scope}(\text{Bool}) \rangle : \text{Bool}
\]

\[
\sim \exists X
\]

if we are satisfied with the standard display form for a scope, or (for example)

\[
\langle \text{exp} \rangle ::= \quad \langle \text{exists} \ \langle \text{scope} \ \langle \text{idl} \ \bar{v} \rangle \ \langle \text{exp} \ P \rangle \ \langle \text{exp} \ Q \rangle \rangle : \text{Scope}(\text{Bool}) \rangle : \text{Bool}
\]

\[
\sim \exists \bar{v}.(P \land Q)
\]

if we are not.

We expect that the bulk of new definitions will define new functions, since these constitute the core of mathematical notations. The abstract syntax of a new function has a very regular form since each component is an \( \langle \text{exp} \rangle \). Consequently we propose an exposition of function definitions that looks more familiar and is based on the function signature, rather than the abstract syntax. Under this plan, the definitions
\textbf{and} : \textit{Bool} \times \textit{Bool} \rightarrow \textit{Bool} \quad \sim p \land q
\textbf{exists} : \textit{Scope(Bool)} \rightarrow \textit{Bool} \quad \sim \exists X

(3.5)

are the condensed form of definitions (3.2) and (3.3). Referring now to (3.5), the name of the production leads the definition. Following the colon are sorts of the arguments separated by the symbol \times. The sorts correspond to the consecutive subexpressions in the right-hand side of the abstract syntax. The sort to the right of \rightarrow is associated with the outer term in the definition. The labels are drawn from the legal identifiers in the display form to the right of the separator \sim and are matched up with subexpressions of the abstract syntax from left to right in lexicographic order. More complex forms that do not fit this simple scheme, such as the customized display form of (3.4), can still be written out in full.

Statements are introduced similarly, but since statement components can be drawn from any of the syntactic categories, they are not candidates for the shorthand used with functions. For example, in type \textit{Bool} we might define a simple one-arm \textbf{if} statement:

\begin{align*}
\langle \textit{stmt}\rangle &::= (\textbf{if} \; \langle \textit{exp} \; B \rangle ; \; \langle \textit{stmt} \; S \rangle ; \; \textit{Stmt}) ; \; \textit{Stmt} \\
&\quad \sim \textbf{if} \; B \; \textbf{then} \; S \; \textbf{fi}
\end{align*}

Statements do not produce values, but we use the sort \textit{Stmt} as the type of a statement component for uniformity and agree that it can be omitted.

We have indicated type correctness rules by relating the sorts of components to the sort of the production. We can add to the expressiveness of the type system by augmenting the constants (sort names \textit{Bool}, \textit{Int}, etc.) of the type expression "language" with variables that vary over all sorts. Sort variables will always be written as script greek letters (\alpha, \beta, etc.), and can be used to construct more flexible signatures that are universally quantified over all sorts. The lexical category of sorts (e.g. \textit{Bool}) is distinct from program variable names, which are lower-case strings. A sort name is a single leading capital followed by one or more lower case letters.

Consider the relational functions \textbf{'}\textbf{=}' and \textbf{'}\textbf{\neq}'. These functions have the same properties, regardless of the type of values they operate over. Equality, for example, is an equivalence relation on \textit{Int} values or \textit{Bool} values or whatever values, and, in usage, is the same function regardless of its argument types. Consequently we only want to define it once, and will need the added flexibility in defining its signature. The signature \textbf{eq} : \alpha \times \alpha \rightarrow \textit{Bool} differs from the signature \textbf{eq} : \alpha \times \beta \rightarrow \textit{Bool} in that the former constrains the argument sorts to be the same. The second is more general, but we choose to restrict the relationals to the former signature, since the extra constraint aids in the inference of types, and the flexibility of the latter confers no additional use (since we never have equality of two values of different type). In general, since type inference has a prophylactic value, constraining the function signatures improves the detection of programming anomalies (which are, in any case, manifest in the proof). On the other hand, overly restrictive signatures reduce the flexibility of the programming notation.

Most remaining syntactic details of a type definition can be extracted from the skeleton definition for type \textit{Bool} in Fig. 1. The identities for predicate logic are given in many texts and are not codified here.

Two statements are introduced in the type \textit{Bool} which are similar to the "guarded command" constructs introduced by Dijkstra in [Dijkstra75]. Both of these constructs are composed of zero or more guarded commands \textit{B} \rightarrow \textit{S}. The intent of \textit{B} \rightarrow \textit{S} is to guard the command \textit{S} with boolean \textit{B} so that \textit{S} can only be started in states satisfying \textit{B}. Using our definitions, we
can write such a guarded command as the block \((\_ \parallel B \parallel S)\), where \_ represents the empty list of identifiers. There are no values to choose for \_ so \(B\) must already be satisfied in order to start \(S\).

**Remark**

The guarded command of Dijkstra is a syntactic construct (in the same sense as our *scope*) and hence has no intrinsic meaning. Our corresponding block \((\_ \parallel B \parallel S)\), on the other hand, is a statement and has the expected meaning defined by the predicate transformer

\[
\begin{align*}
wp.(\text{true} \parallel \text{choose} \_ \parallel B \parallel S',R) & \quad \text{by (2.1)} \\
= wp.(\text{true} \parallel \text{choose} \_ \parallel \text{so} B \parallel S',R) & \quad \text{by (2.14)} \\
= wp.('choose' \_ \parallel \text{so} B' \parallel wp.(S,R)) & \quad \text{by (2.9)} \\
= \exists(\_ \parallel \text{true} \parallel B) \land \forall(\_ \parallel B \parallel wp.(S,R)) & \quad \text{by (2.10)} \\
= B \land wp.(S,R) & \quad \text{pred. calculus} \\
= B \land wp.(S,R) & \quad \square
\end{align*}
\]

The alternative and repetitive statements in [Dijkstra75] are defined over a set of guarded commands \((B_i \rightarrow S_i)\). We will give the same definitions, but, since a guarded command is an
instance of a block, our definitions can be generalized over a set of blocks \((v_i \cup B_i \cup S_j)^i\). This allows us to make a private set of variables \(v_i\), with values that satisfy \(B_i\), available to \(S_j\). Such an extension is not without precedent — variables can appear in CSP guards [Hoare78] to capture the value of a communication as a side effect of a condition that becomes true when the communication is complete. The predicate transformers for the two statements are generalizations of the familiar definitions given in [Dijkstra75] and are given in the Appendix.

3.1.2. Parameterized type definitions

Many programming problems are phrased in terms of collections of data. The data types that are suited to these problems are aggregates like sets and sequences of values. We turn now to the definition of aggregate-value types. Consider, for example, the construction of \(Seq\), a type definition for sequences of values. Amongst the operations will be the familiar index function — for a non-negative index \(i\) and sequence \(t\), the value \(t_i\) is element number \(i\) in the sequence \(t\). What can we give as the signature of this function? If the sequence is heterogeneous (containing values of differing types), the result could be of any type, which we would characterize with the signature \(\text{index} : \text{Seq} \times \text{Int} \to \alpha\). However, there are few operations one could undertake on a value of arbitrary type, so it is not unreasonable to restrict ourselves to homogeneous aggregates: sequences and sets in which all elements have the same type.

Remark

We can always gather any finite collection of values of differing types in a tuple value, on which the analog of the index operation, \text{tuple-extr} (projection), can be used to select elements from fixed components. \(\square\)

For a sequence of type \(\text{Bool}\), say, the index function would have signature \(\text{index} : \text{Seq} \times \text{Int} \to \text{Bool}\), while for a sequence of integers it would be \(\text{index} : \text{Seq} \times \text{Int} \to \text{Int}\). There still is no single signature for the indexing operation, but now it is parametric in the type of the sequence elements. We could give several different type definitions, i.e. \(\text{BoolSeq}, \text{IntSeq}\), etc., that fix individual signatures, but this would lead to many different index functions that all behave identically. Since it is our intention to organize a type definition around a set of properties, we extend the type expression language further to permit type definitions to be parametric over sort variables. To accommodate our simple type inference system, these will be the only parameters permissible in a type definition. This, for the time being, prohibits type definitions like \(\text{IntMod}(N)\), the integers \(mod\ N\), but the extension to include them is straightforward. For sequences and sets we can now write type definitions \(\text{Seq}(\sigma)\) and \(\text{Set}(\sigma)\); the latter is presented in Fig. 2. The parameter in a parameterized sort is not an argument-passing mechanism in a type definition; it denotes a class of definitions.

The set of functions defined in Fig. 2 is by no means complete, but is considerably larger than the minimum necessary for a development of a theory of sets [Manna85]. It is intended to correspond more generally to the functions introduced in a specification language like Z [Hayes86, Spivey86]. The meaning of the less familiar functions is given in the Appendix.

With regard to the display forms, note that the empty set and the subset relation each have two different display forms for the same function. This is a matter of convenience for notations. The converse situation, one display form for two different functions, is a superficial form of overloading and is more subtle. In Fig. 2 this situation occurs with the definitions for \text{singleton},
\begin{figure}[h]
\begin{center}
\begin{tabular}{|l|c|}
\hline
\textbf{type} & \textit{Set}(\sigma) \\
\hline
\textbf{exp} & \\
\hline
null & \textit{Set}(\sigma) \quad \sim \{\}\text{ or } \emptyset \\
singleton & \sigma \rightarrow \textit{Set}(\sigma) \quad \sim \{a\} \\
member & \sigma \times \textit{Set}(\sigma) \rightarrow \textit{Bool} \quad \sim a \in s \\
union & \textit{Set}(\sigma) \times \textit{Set}(\sigma) \rightarrow \textit{Set}(\sigma) \quad \sim s \cup t \\
interesection & \textit{Set}(\sigma) \times \textit{Set}(\sigma) \rightarrow \textit{Set}(\sigma) \quad \sim s \cap t \\
difference & \textit{Set}(\sigma) \times \textit{Set}(\sigma) \rightarrow \textit{Set}(\sigma) \quad \sim s \setminus t \\
complement & \textit{Set}(\sigma) \rightarrow \textit{Set}(\sigma) \quad \sim \overline{s} \\
construct & \textit{Scope}(\sigma) \rightarrow \textit{Set}(\sigma) \quad \sim \{X\} \\
reduce & (\sigma \times \sigma \rightarrow \sigma) \times \textit{Set}(\sigma) \rightarrow \sigma \quad \sim f/s \\
each & (\beta \rightarrow \sigma) \times \textit{Set}(\beta) \rightarrow \textit{Set}(\sigma) \quad \sim f"s \\
seqtoset & \textit{Seq}(\sigma) \rightarrow \textit{Set}(\sigma) \quad \sim \{t\} \\
subset & \textit{Set}(\sigma) \times \textit{Set}(\sigma) \rightarrow \textit{Bool} \quad \sim s \subseteq t \text{ or } t \supseteq s \\
size & \textit{Set}(\sigma) \rightarrow \textit{Int} \quad \sim \#s \\
\hline
\end{tabular}
\end{center}
\caption{Type Definition \textit{Set}(\sigma)}
\end{figure}

\textit{construct} and \textit{seqtoset}. The type inference system we will be considering next is not capable (in the form presented here) of solving for the unknown types in the presence of multiple signatures for the same function. The only form of overloading that can be accommodated is \textit{parametric} — when all signatures can be expressed as instances of a single type expression involving free type variables. Consequently for the situation presented here, we have to rely on the fact that the abstract syntax separates the three different signatures, so that, in essence, we require the distinction between the cases be made during program development.

### 3.2. Type Inference System

We have attributed the abstract syntax with type expressions that relate the type of a term to the types of its subterms. Given a term \( T \) in \( G \), the type of the complete term \( T \) can be related by these type expressions to the types of \( \nu \), \( T \), the free occurrences of identifiers in \( T \). The type inference system determines the most general such relation and fails if the relation is empty. The relation is developed by synthesizing an ever more restrictive relation upwards into the syntax tree of \( T \), starting with unconstrained relations at the \( \langle id \rangle \) leaves.
Definition (Type expression)

A type expression is (a) a free variable \((\alpha, \beta, \ldots)\) that ranges over type expressions, or (b) an unparameterized sort, or (c) a parameterized sort with a type expression in each parameter position. We will let capital greek letters \((\Sigma, \Pi, \Phi, \Psi, \ldots)\) represent type expressions. A type expression denotes the set of all substitutions of sorts for variables. For type expressions \(\Sigma, \Psi\) and type variable \(\tau, \Sigma_\Psi\) is a type expression, but denotes a class of sorts that can be no larger than \(\Sigma\). The unparameterized sorts that we have mentioned up till now are \(\text{Bool, Int and Stmt}\) for \((\text{stmt})\) terms. The parameterized sorts we have mentioned are \(\text{Set(\tau)}, \text{Seq(\tau)}\) and \(\text{Scope(\tau)}\) for the sort of a scope with range type \(\tau\). Two more parametric sorts are \(\text{Func(\alpha, \beta)}\) and \(\text{Tuple(\sigma_0, \ldots, \sigma_{n-1})}\) (which may have a varying number of parameters). For clarity we will write these sorts as \(\alpha \rightarrow \beta\) and \(\sigma_0 \times \cdots \times \sigma_{n-1}\).

Definition (Type bindings)

A type binding \(A\) is a function from identifiers to type expressions. The domain of \(A\) is a subset of all identifiers. The type binding \((A, x : \Sigma)\) extends \(A\) to associate \(x\) with type expression \(\Sigma\); it requires that \(x \in \text{dom.} A \Rightarrow (A, x = \Sigma)\), i.e. \(x\) is undefined in \(A\) or is already defined to the identical expression. The singleton type binding \(x : \Sigma\) is undefined everywhere except at \(x\), and the binding \(\overline{x} : \overline{\Sigma}\) stands for \(x_0 : \Sigma_0, \ldots, x_{n-1} : \Sigma_{n-1}\).

Definition (Type formula)

A type formula has the form \(A \vdash T : \Sigma\) where \(A\) is a type binding and \(T\) is any term in the abstract syntax generated by \(G\). The type formula expresses a relation between the types of identifiers and the type of \(T\). For example, the type formula \(s : \text{Set(\sigma)}, x : \sigma \vdash s \cup \{x\} : \text{Set(\sigma)}\) expresses that, for any choice of type for \(\sigma\), if \(s\) has type \(\text{Set(\sigma)}\) and \(x\) has type \(\sigma\), then \(s \cup \{x\}\) will be of type \(\text{Set(\sigma)}\).

We now give a formal system of axioms and rules of inference that defines valid type formulas. If \(T\) is a term and the formal system can be used to deduce \(A \vdash T : \Sigma\) for some \(A\) and \(\Sigma\), then \(T\) is well-typed and \(A\) contains the type constraints on \(fv.T\). We start with inference rules to manipulate type formulas.

\[
\begin{align*}
\text{\[EXT\]} & \quad A \vdash T : \sigma \quad \text{for } x \not\in \text{dom.} A \\
& \quad A, x : \alpha \vdash T : \sigma \\
\text{\[COMB\]} & \quad A \vdash T_0 : \sigma_0, \ldots, A \vdash T_{n-1} : \sigma_{n-1} \\
& \quad A \vdash T_0 : \sigma_0, \ldots, T_{n-1} : \sigma_{n-1} \\
\text{\[SUB\]} & \quad A \vdash T : \Sigma \\
& \quad A_\Psi \vdash T : \Sigma_\Psi
\end{align*}
\]

Now for each production of \(G\) we add a rule of inference with a name taken from the production name. The \(\langle id \rangle\) production is associated with an axiom \(id\) that introduces the identifier as a term with an unconstrained type binding; this is an axiom that is used to deduce type of an expression from the types of its components. The \(\langle idl \rangle\) production corresponds to an axiom that introduces identifier(s) with an unconstrained type binding but without a typed term; this is the axiom that will be used when new identifiers are introduced in a scope.
[id] \[ x : \sigma \vdash x : \sigma \] for any \( x, \sigma \)

[idl] \[ \bar{x} : \bar{\sigma} \vdash \] for any \( \bar{x}, \bar{\sigma} \)

Each production of \langle exp \rangle or \langle stmt \rangle in a type definition contributes an inference rule. The abstract syntax

\[
(f \ W_0 : \Psi_0 \ \cdots \ W_{n-1} : \Psi_{n-1}) : \Psi
\]

in a type definition gives rise to the inference rule

\[
\begin{align*}
A & \vdash W_0 : \Psi_0, \ \cdots, W_{n-1} : \Psi_{n-1} \\
A & \vdash (f \ W_0 \ \cdots \ W_{n-1}) : \Psi
\end{align*}
\]

or, if \( n=0 \),

\[
\begin{align*}
& \vdash (f) : \Psi
\end{align*}
\]

Now we give inference rules for the basic \langle exp \rangle in \( G \) introduced in chapter 2 that are not part of any type definition. These are: function construction and application and tuple construction and projection.

[t-cons] \[
A \vdash e_0 : \sigma_0, \ \cdots, e_{n-1} : \sigma_{n-1} \\
A \vdash (e_0, \ \cdots, e_{n-1}) : (\sigma_0 \times \ \cdots \times \sigma_{n-1})
\]

[t-extr] \[
A \vdash e : (\sigma_0 \times \ \cdots \times \sigma_{n-1}) \\
A \vdash e.i : \sigma_i
\]

[f-appl] \[
A \vdash f : \alpha \to \beta, \ e : \alpha \\
A \vdash f.e : \beta
\]

[f-abs] \[
\bar{v} : \bar{\sigma} \vdash P : Bool, \ e : \tau \\
\vdash \lambda(\bar{v} \parallel P \parallel e) : \sigma_0 \times \ \cdots \times \sigma_{n-1} \to \tau
\]

Productions from Chapter 2 that treat the interactions between identifiers, expressions and statements make up the remaining rules.

[assign] \[
A, \nu_0 : \sigma_0, \ \cdots, \nu_{n-1} : \sigma_{n-1} \vdash e : (\sigma_0 \times \ \cdots \times \sigma_{n-1}) \\
A, \nu_0 : \sigma_0, \ \cdots, \nu_{n-1} : \sigma_{n-1} \vdash \nu_0, \ \cdots, \nu_{n-1} := e : Stmt
\]

[compose] \[
A \vdash S_0 : Stmt, \ \cdots, A \vdash S_{n-1} \\
A \vdash S_0; \ \cdots; S_{n-1} : Stmt
\]

[scope] \[
A, \bar{v} : \bar{\sigma} \vdash P : Bool, W : \tau \\
A \vdash (\bar{v} \parallel P \parallel W) : Scope(\tau)
\]

[choose] \[
A, \bar{v} : \bar{\sigma} \vdash P : Bool \\
A, \bar{v} : \bar{\sigma} \vdash \text{choose } \bar{v} \text{ so } P : Stmt
$$[\text{valblk}] \quad A, \overline{\nu} : \Phi \vdash P : \text{Bool}, S : \text{Stmt}, E : \sigma$$
$$A \vdash (\overline{\nu} \parallel P \parallel S \text{ yields } E) : \sigma$$

$$[\text{block}] \quad A \vdash W : \text{Scope(Stmt)}$$
$$A \vdash W : \text{Stmt}$$

These rules allow us to deduce the most general valid type formula $A \vdash T : \Sigma$ for an arbitrary fragment $T$, but they also allow us to deduce any substitution instance of the most general formula. Clearly, indiscriminate use of rule $[\text{SUB}]$ leads to a less general valid type formula. We give an algorithm $M$ that incorporates the unification algorithm of Robinson [Robinson65, Milner78] to implement deduction in this formal system that finds the most general valid type formula for $T$ (when it exists).

Algorithm $M$ works on the abstract syntax tree of $T$. Starting at the leaves, $M$ can always apply one of $[\text{id}], [\text{idl}]$ and $[f_0]$. The resulting type formula is the most general because it makes no restrictions in the first two cases, or because it is the signature of a function introduced in the notation. Now consider the operation of $M$ inductively on some term $T$. Let $T = (f \ W_0 \cdots W_{n-1})$ and assume that $M$ has developed the most general type formulas for each of the subproductions:

$$A_0 \vdash W_0 : \Sigma_0, \cdots, A_{n-1} \vdash W_{n-1} : \Sigma_{n-1}$$

Starting from this point, $M$ applies the $[\text{EXT}]$ rule to make the domains of the $A_i$ equivalent using fresh type variables for any new variables that extend a type binding. This preserves generality since the type variables of the identifiers added to $A_i$ do not appear in $\Sigma_i$. Then $M$ ensures that the type variables in the $A_i$ are mutually disjoint by straightforward relabeling of the $A_i$. This does not alter the generality of the type formulae. Now $M$ attempts to unify the type bindings $A_i$ with Robinson's algorithm, which finds substitutions $\Pi_i$ for the free type variables $\pi_i$ in each $A_i$ such that

$$A' = [A_0]_{\Pi_0}^{\pi_0} = \cdots = [A_{n-1}]_{\Pi_{n-1}}^{\pi_{n-1}} \quad (3.6)$$

The substitution can be carried out with rule $[\text{SUB}]$, and it is the most general, i.e. any other substitution that satisfies (3.6) is an additional substitution on the $\Pi_i$. This is guaranteed by Robinson's algorithm. This is also the substitution under which all the individual type constraints in the $W_i$ are simultaneously met. Now $M$ uses $[\text{COMB}]$ to consolidate all the subproduction types under one type binding, yielding

$$A' \vdash W_0 : [\Sigma_0]_{\Pi_0}^{\pi_0} \cdots W_{n-1} : [\Sigma_{n-1}]_{\Pi_{n-1}}^{\pi_{n-1}} \quad (3.7)$$

Finally, $M$ unifies the type expressions on the right hand side of (3.7) with the antecedent of inference rule $[f]$ to obtain the most general substitution that equates the $\Psi_i$ and the $[\Sigma_i]_{\Pi_i}$ to obtain a substitution to apply to $A'$ and $\Psi$ to yield $A''$ and $\Psi'$.

Now $A'' \vdash (f \ W_0 \cdots W_{n-1}) : \Psi'$ is a valid type formula since it was obtained by the rules of inference from valid type formulae, and is the most general such formula because each step in its construction preserved its generality.

This completes the explanation of the inductive step of $M$. If either of the unification steps fail, there is no consistent type binding for $T$.

Example.
Consider the assignment \( x, y := (y, x) \). In the abstract syntax this corresponds to

\[
(assign \ (idl \ x \ y) \ (t-cons \ (id \ x) \ (id \ y)))
\]

For this statement the inference algorithm will obtain the typing

\[
x : \alpha, y : \alpha \vdash x, y := (y, x) : Stmt
\]

(3.8)

To demonstrate how this was achieved, we start with the innermost terms, identifiers \( x \) and \( y \), individually unrestricted:

\[
x : \alpha \vdash x : \alpha \quad \text{and} \quad y : \beta \vdash y : \beta
\]

Now we move through a single inductive step of \( M \) to construct the tuple:

\[
\frac{x : \alpha \vdash x : \alpha \quad y : \beta \vdash y : \beta}{x : \alpha, y : \beta \vdash x : \alpha \quad x : \alpha, y : \beta \vdash y : \beta} \quad [\text{EXT}]
\]

\[
\frac{x : \alpha, y : \beta \vdash x : \alpha \quad x : \alpha, y : \beta \vdash y : \beta}{x : \alpha, y : \beta \vdash (y, x) : \beta \times \alpha} \quad [\text{COMB}]
\]

The \( \langle idl \rangle \) branch of the assignment is simple:

\[
x : \alpha, y : \beta \vdash \ [idl]
\]

In the last step we have to specialize the type bindings because \([assign]\) requires type agreement between old and new values for variables.

\[
\frac{x : \alpha, y : \beta \vdash (y, x) : \beta \times \alpha \quad x : \alpha, y : \beta \vdash}{x : \alpha, y : \beta \vdash (y, x) : \beta \times \alpha \quad x : \alpha, y : \beta \vdash (y, x) : \alpha \times \alpha} \quad [\text{COMB}]
\]

\[
\frac{x : \alpha, y : \beta \vdash}{x : \alpha, y : \beta \vdash x, y := (y, x) : Stmt} \quad [\text{assign}]
\]

Thus the swap is type correct whenever \( x \) and \( y \) have the same type.

Remark

The invariance of the type of identifiers throughout their scope is a consequence of the \([assign]\) and \([compose]\) rules and is purposefully present in the type system. In fact, from a logical point of view there is nothing wrong with a variable that assumes values of different types within its scope. We could not argue that we restrict the type of an identifier to be constant over its lifetime for efficiency reasons: an implementation module might provide a more attractive implementation for a variable with static storage, but that is where the restriction should come into force, not at the level of definition. Probably the best reasons to cite for our choice lie along the same lines of our choices for signatures for equality and indexing: the type system has more opportunity to catch errors when more constraints have to be satisfied. \( \square \)

Thus far we've talked about program fragments independent of the assertions in the correctness proof. Recall that, at least syntactically, assertions behave like \( \text{Bool} \) expressions, and therefore a simple extension can be made to include the annotation in type inference:
\[ \frac{A \vdash P : \text{Bool}, Q : \text{Bool}, S : \text{Stmt}}{A \vdash (P \quad S \quad Q) : \text{Stmt}} \]

Such a rule makes a difference. For the swap of example (3.8) we obtained a typing \( x, y : \sigma \). Now consider the same statement annotated with

\[ \{ x \land y \} \quad x, y := (y, x) \quad \{ x \land y \} \]  \hspace{1cm} (3.9)

Type inference assigns \( x, y : \text{Bool} \) to the expressions, so the annotated swap is only type correct for \( x \) and \( y \) boolean. It is important to include the assertions in the type inference process because one function of the system is to eliminate the need for type predicates in the program proof. In the example above the pre- and post- condition are the same, and the reader might be convinced of the formal correctness. The weakest precondition is actually \( y \land x \), and the equivalence \( x \land y \equiv y \land x \) given by (3.1) is contingent on \( \text{Bool} \cdot x \land \text{Bool} \cdot y \). This is precisely what the type inference system has restricted (3.9) to.

We close this section with a few thoughts about the meaning of valid type formulae that contain free type variables. First, by comparison of the definition of \( \text{fv} \) (2.2 (a)-(e)) in section 2.2.6 with the inference rules, it can be observed that if \( A \vdash T : \Sigma \) is a valid formula, then \( A \) contains bindings for exactly and only the variables that occur free in \( T \). If \( \Sigma \) contains type variables that do not appear in \( A \), then \( T \) is undetermined by \( A \); the type of \( T \) is independent of the types of the free variables. An undetermined type formula can arise only from an undetermined signature like \( \text{null} : \to \text{Set}(\alpha) \) and is not problematic as an intermediate type formula. If the type bindings in \( A \) contain any type variables then \( T \) is polymorphic. If \( T \) is not undetermined and the type bindings in \( A \) contain no type variables then \( T \) is monomorphic.

Suppose we have a program \( S \), with \( \text{fv} \cdot S = \{ x \} \), and the type inference system obtains the type formula \( x : \Sigma \vdash S : \text{Stmt} \), where \( \sigma \) is a free type variable in the type expression \( \Sigma \). The interpretation is that \( S \) is type correct under a variety of possible choices of type for \( x \). The term polymorphic is often used to describe a function (or program) that can operate on several different types of values. However, in ML, where the type inference system of Milner is used, polymorphism has a narrower meaning: a function (or program) is polymorphic if its signature contains free type variables \textit{and} its executable code is invariant under change of type (and does not employ run-time type analysis). In ML, an element of arbitrary type \( \sigma \) can be prepended to a list of elements of the same type without examination of a run-time type tag, because the value is referenced via a pointer and prepending involves the same \( \text{CONS} \) operation to the list of other pointers, regardless of the type of values in the list. But the equality relation, for example, is not polymorphic in the ML sense, since, in general, it must use code dependent on the type of values (e.g. sets or integers). So while the ML list operations have signatures identical to the corresponding operations in our definition of \( \text{Seq}(\sigma) \) (see the Appendix), the ML equality relation does not, because its implementation depends on the type of its values.

Our signatures reflect the proof-theoretic properties of a function, so that we say that the \textit{proof} of \( S \) is invariant under change of type. It does not mean that there is an implementation of \( S \) which works for all types of \( x \), although this is not precluded. For example, one implementation module we would define is an implementation for sequences restricted to (LISP) list operations. This implementation \textit{would} have the property that it was a (partial) implementation of \( \text{Seq}(\sigma) \) without referring to \( \sigma \). But in general the implementation of \( S \) will depend on the types chosen for \( x \), which need to be instantiated. This makes \( S \) a \textit{generic} program.
Type signatures in type definitions need not be limited to those that satisfy ML-style polymorphism, since that is an implementation issue. Both flexibility of notation and efficiency can be compromised by insisting on type-independent implementations.

The type inference algorithm we employ can use the more general generic signatures of operations to record a generic type-correctness proof. A program with a generic type can have an implementation that is invariant over the instances of the type, or it can have an implementation that varies with the context in which it is instantiated.
4 Implementations: Modules and Directives

4.1. Introduction and Overview

In this chapter we will discuss implementation modules and implementation directives. We assume now that we have constructed a full programming notation with the mechanisms and approach described in chapters 2 and 3. An implementation module provides an implementation for a subset or a restriction of the full notation. A variety of implementation modules will be collected into a library. An implementation directive specifies an implementation from such a library to be used to translate the program to a more efficient form or even into a different programming notation.

The general form of an implementation module shares some of the features of a type definition, but its purpose is quite different. Whereas each type definition uniformly generates a new class of expressions and statements, an implementation module identifies an (irregular) subset of all expressions and statements. The subset can include expressions or statements that span several type definitions, and it is identified syntactically as the set of expressions, scopes and statements from all combined type definitions that match a collection of templates. This subset can be further qualified by semantic restrictions in the form of a predicate or precondition with each template that restricts the set of states in which matched expressions or statements are associated with an implementation.

Typically a module will identify a set of expressions and statements that operate on one or more common variables. The module then indicates how the values denoted by these variables are represented, and gives, for each expression and statement in the set, corresponding operations on that representation.

A module presenting implementations for all operations of a single type on a single common variable provides roughly the same facilities as a SIMULA class [Dahl70] (which implements a number of operations on a single variable). An implementation module is a proper generalization of a SIMULA class, however, since it is possible to provide implementations for only some of the operations of a given type as well as to provide implementations for related operations on values of a different type. In addition, an implementation module can provide a single implementation for several related variables, whereas a SIMULA class always implements operations on a single variable. As an example of both these extensions, consider the expression \(#s + #t\) (the sum of the number of elements in set \(s\) and set \(t\)). A module that implements both \(s\) and \(t\) can provide an implementation for the entire expression, despite the fact that (1) the expression is not an operation exclusively associated with \(s\) or \(t\) (this is the source of the vexing asymmetry in object oriented languages), and (2) the outermost operation (+) is not an operation on sets. Implementation of this expression might be desirable, for example, when the values in \(s\) and \(t\) can be represented together so that determination of the individual set sizes is expensive or impossible but determination of the sum of their sizes is not. A SIMULA class providing (the same) implementations for \(s\) and \(t\) could only implement \(#s\) and \(#t\), relying on an
implementation of integers to yield their sum.

A module presenting implementations for all operations of a single type without reference to a particular variable is analogous to an abstract data type implementation facility like the CLU cluster, the Modula 2 module, or the ADA package. Such facilities are also generalized by implementation modules, since the module need not implement all the operations of a type and can implement expressions involving several operations of several types.

As an example of the difference in approaches, an iterator is easily implemented using our module approach but is difficult to do with the "operation-for-operation" implementation approaches. Consider a set $s$ and the class of statements described by

\[ \texttt{foreach}(v \in s \rightarrow (\texttt{stmt} S)) \]

This class describes repetition of an arbitrary statement $S$ with each member $v$ of the set $s$. We can easily describe an implementation of this particular class of statements in terms of the representation of $s$ and the implementation of $S$. For example, if $s$ is represented by the sequence $c(0..k)$ and $S$ is implemented as $S'$, a possible implementation is:

\[ (i \in 0..k \rightarrow (v := c_i \rightarrow S')); i := i+1 \texttt{od} \]

With a SIMULA class this sort of iterator must be implemented by augmenting type set with access operations like procedures $s$.init_iter($v$), $s$.next_value($v$) and a function $s$.iter_done(), and rewriting the iterator into an equivalent loop employing these operations. With abstract data types (ADTs) such access operations can not even be formulated because they have internal state. Similar problems arise with an ADT implementation of an expression like $\langle \exp f \rangle / s$; this expression would have to be implemented in the data type that defined the (second order) reduce operation, in which case it can not take advantage of a particular representation for $s$.

In order to maximize the utility of a module, we want to restrict only some parts of a notation, but not others. For example, we would like to produce an implementation of sets that uses a heap representation to organize the members of a set, independent of their type. Implementations of parameterized types that are independent of the implementations of the parameters are generic. We will want to construct generic implementation modules because such modules are good candidates for a library of general and reusable implementations.

We begin the discussion of implementation modules with an explanation of the form and meaning of templates used to identify a restricted notation.

### 4.2. Patterns and Templates

A pattern is a sentence of the abstract syntax that can contain nonterminals. A pattern matches a sentence of the abstract syntax containing only terminal symbols if the sentence can be derived from the pattern. We will specialize these definitions to our setting: for $P$ a production of $G$, a $P$–pattern is a sentence $T$ over terminals and nonterminals ($\langle \texttt{stmt} \rangle$, $\langle \texttt{scope} \rangle$, $\langle \exp \rangle$ and $\langle \texttt{id} \rangle$) of $G$ such that

\[ P \rightarrow_G T \]

The pattern $T$ matches all sentences $X$ composed only of terminals such that

\[ T \rightarrow_G X \]
Just as a program is the display form of its abstract syntax, we will define a \textit{P-template} to be the display form of a \textit{P-pattern}, with the (standard) understanding that the display form of a nonterminal is the name of the nonterminal enclosed in angle brackets. Generally speaking, when we say a template matches a program fragment, we mean that the pattern corresponding to the template matches the abstract syntax subtree corresponding to the program fragment.

We also add two context-sensitive restrictions that may be used in patterns to limit the program fragments matched.

First, recall that in the discussion on the basic components of a programming notation (section 2.2.1), we introduced “labels” in nonterminal productions to vary over (the text of) their yield. We now require that the sentences matched by a pattern lead to a consistent instantiation of the labels: all nonterminals with the same label must derive the same sentence. Thus a repeated label restricts the set described by the unlabeled pattern. The template $\langle \text{exp } t \rangle + \langle \text{exp } t \rangle$, for example, only matches additions whose right and left side are identical expressions. A label can only be repeated with multiple occurrences of the same nonterminal (this precludes $\langle \text{stmt } S \rangle$ and $\langle \text{exp } S \rangle$ occurring in the same template). Identifiers that occur as part of a template must be distinct from the labels of nonterminals (this precludes $s + \langle \text{exp } s \rangle$).

A pattern $T$ may be turned into a sentence $T_{\text{sent}}$ by replacing all $\langle \text{exp} \rangle$ and $\langle \text{id} \rangle$ nonterminals by their labels and by replacing each $\langle \text{stmt} \rangle$ non-terminal with the null statement $\text{skip}$. The sentence $T_{\text{sent}}$ is then composed solely of terminals of $G$, and is matched by $T$, since the labels (which are identifiers) can be derived from $\langle \text{exp} \rangle$ and $\langle \text{id} \rangle$ and $\text{skip}$ can be derived from $\langle \text{stmt} \rangle$.

The second restriction concerns the type and implementation of the expressions and statements matching nonterminals in a pattern. In the construction of type definitions (section 3.1.1) we annotated nonterminals with type expressions to define a portion of the signature of the operation being introduced. Since all operations in a pattern have already been introduced by type definitions, the type of each nonterminal of $T$ can be obtained from sentential form $T_{\text{sent}}$ with type inference algorithm $M$.

The inferred type of a nonterminal is at least as general as the inferred type of any matching program fragment because such a fragment can only restrict the typing of the identifier taking its place in $T_{\text{sent}}$. For example, if we let $T$ be the pattern $\langle \text{exp } v \rangle = \langle \text{exp } v \rangle$, then $T_{\text{sent}}$ is $v = v$. The inferred type for $T_{\text{sent}}$ is $v : \sigma \vdash (v = v) : \text{Bool}$, while any other fragment matched by $T$, such as $5 = 5$, is a substitution instance of $\sigma$, in this case $\text{Int}$.

The program fragments matched by a nonterminal can be limited by attaching the name $M$ of an implementation. This restricts the matched text to terms for which $M$ provides an implementation. For example, if $P_{\text{INT}}$ is an implementation of $\text{Int}$ values, then the pattern

\[ a : = a \cup \{(\text{exp } e) : P_{\text{INT}}\} \]

restricts the possible matches for $e$ to expressions implemented by $P_{\text{INT}}$. If the name attached to a nonterminal implements values of a type that is not a substitution instance of the type expression inferred from $T_{\text{sent}}$, then $T$ is a type-incorrect pattern.

### 4.3. Implementation Module Syntax

Sample implementation module BITV in Fig. 3 serves to motivate a description of implementation module syntax. The module represents a set of integers as a bitvector using a PASCAL boolean array and provides PASCAL implementations of a few operations on the set.
module BITV (n : P_INT) impl
  a : Set(Int)
  ~ c : array [0..n'-1] of Integer

coupling
a = \{ i \mid 0 \leq i < n \land c(i) \parallel i \}

stmt
  \{ true \} (\textsf{a} := \{\} : \textsf{P_STMT}
  ~ \text{for } i := 0 \text{ to } n'-1 \text{ do } c(i) := \textsf{false}
  \{ 0 \leq e < n \} (\textsf{a} := \textsf{a} \cup (\langle \textsf{exp} e \rangle : \textsf{P_INT}) : \textsf{P_STMT}
  ~ c(e') := \textsf{true}

exp
  \{ 0 \leq e < n \} (\langle \textsf{exp} e \rangle : \textsf{P_INT} \in \textsf{a}) : \textsf{P_BOOL}
  ~ c(e')

Fig 3. BITV implementation module.

Although the implementation module shares some of the structure of a type definition, there are differences.

- An implementation module has a more detailed interface than a type definition, which only provides parameterization by type variables.

- A type definition can give the meaning of operations axiomatically, in the props section of the definition. There is no such section in an implementation module since the semantics of all operations that appear within are already fixed by the relevant type definitions.

- A type definition adds new \langle exp \rangle and \langle stmt \rangle productions; an implementation module selects instances of these productions via exp– and stmt– templates. Although no new scope productions are introduced with type definitions, an implementation can also select instances of a scope with scope–templates. Each template is associated with an implementation.

- The implementation module gives a coupling between abstract and concrete variables and includes, for each template, a precondition governing the states in which the implementation is adequate.

The four main components of an implementation module are, in order, the interface, the coupling, zero or more definitions, and zero or more implementation units. Refer to the general outline of an implementation module in Fig. 4 with the description of the syntax of the components that follows.

Interface.

The interface adapts the meaning of a module with arguments supplied as part of an implementation directive. For example, it can be used to fix the size of a concrete representation or instantiate an implementation within a generic module. The identifier \text{M} is the name of the module and is lexically distinct from type names and program variables; it is completely capitalized.
The parameters $\bar{G}$ vary over implementations. A generic implementation module has one or more of these parameters; the sample module BITV is not generic. The parameters $\bar{n}$ vary over values in the notation, although each $n_i \in \bar{n}$ may optionally be constrained to values represented by a corresponding module $N_i$. For example, any argument for parameter $n$ in module BITV must be implemented by module P_INT.

The identifier(s) $\bar{a}$ represent the set of variables in the program text to which the implementation directive is attached. The implementation units identify a collection of terms involving $\bar{a}$. The $\bar{a}$ are known as abstract or program variables. Type expression $\Sigma$ identifies the type(s) of the abstract variable(s) $\bar{a}$ that are implemented by $M$; it must be a substitution instance of the typing for $\bar{a}$ inferred from the implementation units; hence it restricts the set of matched terms. If $\Sigma$ contains free type variables, then $M$ is prepared to implement a class of possible types of $\bar{a}$; in this case $M$ is a generic implementation.

The concrete or representation variables $\bar{c}$ are introduced after the separator "$\dashv$" to represent the values of $\bar{a}$. Type expression $\Pi$ defines the type of the concrete variables. If $M$ is generic, $\Pi$ can refer to type variables of $\Sigma$. If $M$ is generic but $\Pi$ contains no type variables, then $M$ is a polymorphic implementation in the ML sense: the implementation is independent of the instantiation of types for the free type variables of $\Sigma$.

**Coupling.**

The coupling is a predicate that relates the values of abstract variables $\bar{a}$ to the values of their representation $\bar{c}$. For convenience we use the name $I$ for the coupling predicate unless specified otherwise. Often more than one coupling is under discussion and so we identify the appropriate coupling by subscripting it with the name of the module where it appears, e.g. $I_M$.

The coupling can refer only to identifiers $\bar{a}, \bar{c}$ and $\bar{n}$ introduced in the interface. The coupling for generic modules can, in addition, refer to the coupling $I_G$ of argument module $G \in \bar{G}$. In some contexts we treat $I$ as parameterized by $\bar{a}$ and $\bar{c}$, i.e. as $\lambda((\bar{a}, \bar{c}) \parallel I)$. The remaining free identifiers $\bar{n}$ and predicates $I_G$ are considered universally quantified at the outermost level in any argument involving $I(\bar{a}, \bar{c})$.

**Definitions.**

A $\langle \text{Defn} \rangle$ has the form $P \ h \equiv T_P$, where $P$ is the name of a production of $G$ (e.g. $exp$), and $T_P$ is $P$–template. The definition simply introduces the name $h$ as an abbreviation for the template or program fragment $T_P$ in subsequent implementation units. For example, the definitions
defn
\(\langle \text{exp min_s} \rangle \equiv \min / \langle \text{exp s} \rangle : \text{HEAP} \)
\(\langle \text{stmt S} \rangle \equiv x := 0\)

mean that in the remainder of the module each free occurrence of \(\text{min_s}\) should be replaced by the template \(\langle \text{exp s} \rangle : \text{HEAP}\) and that each free occurrence of \(S\) should be replaced by the statement \(x := 0\). The syntactic category is required on the left hand side in order to parse the right hand side. Since \(\text{min_s}\) defines a template, \(\text{min_s}\) can only appear wherever a template can appear while \(S\) can appear anywhere in the module that a \(\langle \text{stmt} \rangle\) is required, either in a template or in the text of an implementation.

In order to leave this "macro" facility as simple as possible no (recursive) substitution is carried out in the replacement text. Consequently a definition like \(id x \equiv \langle id x \rangle\) is reasonable and indeed quite convenient. With such a definition the identifier \(x\) can take the place of a non-terminal in subsequent templates. For example, the template \(\text{choose \langle id x \rangle} \text{ so } \langle id x \rangle \in s\) can be written as \(\text{choose } x \text{ so } x \in s\) following such a definition.

Implementation Units.

Each implementation unit has the form \(\langle \text{class} \rangle \{P\} T_{\text{class}} : N \sim W\). The \text{class} is one of \text{scope}, \text{exp}, or \text{stmt} and indicates the kind of template and implementation that follow, as well as where in a program text the template applies (scopes, expressions or statements, respectively). If an implementation module does not provide any \text{scope} templates, then it will only be possible to use the implementation with variables that occur free in a program; these are the only variables that can be used without being introduced in a scope. The template \(T_{\text{class}}\) identifies a collection of terms, namely those matched by it, that are implemented by \(W\).

The identifier \(N\) is the name of the module that couples the abstract value of a program fragment matched by \(T_{\text{class}}\) to its representation. For example, in module BITV the expression \(e \in a\) is a boolean value that is coupled to its implementation \(c(e')\) by module P_BOOL. The module name \(N\) may be \(M\) (the module under definition), one of the module parameters \(G\), or a known module (such as P_BOOL). If the type of template \(T_{\text{class}}\) is \(\Sigma\) and \(N\) is \(M\), then \(N\) can be omitted.

New identifiers \(\bar{v}\) can be introduced as labels in \(T_{\text{class}}\) or as parameters of parameterized modules that may appear in \(N\) or as part of a nonterminal in the template. The scope of \(\bar{v}\) is limited to the implementation unit in which they are introduced. Precondition \(P\) makes a further semantic restriction on the possible starting states in which the implementation applies and is stated in terms of \(\bar{a}, \bar{n}\) and \(\bar{v}\). Note that the precondition cannot refer to \(\bar{c}\), thereby ensuring that the precondition is a predicate over client variables and terms and in the same domain of discourse as the client program.

The implementation \(W\) can refer only to representation variables, but the only such variables so far are the \(\bar{c}\) representing the \(\bar{a}\). Any abstract variable \(v_{\bar{v}}\) has a corresponding representation \(v'\). If the module implementing \(v\) is fixed by the template, the type of \(v'\) can be obtained from that module. For example, the template

\[a := a \cup \{\langle \text{exp } v \rangle: \text{P_INT}\}\]

requires \(v\) to be implemented by module P_INT (given in Fig. 5) where \(v'\), the representation of \(v\), is of PASCAL type \text{Integer}, and so can meaningfully be used in the implementation.
\( c(\nu') := true \) associated with the template above. For a nonterminal \( \nu \) whose implementation is not fixed by the template, we take the type and implementation of \( \nu' \) to be the same as the type and implementation of \( \nu \).

When a list of concrete variables \( c_0, \ldots, c_{n-1} \) is used to represent an abstract variable \( a \), the individual concrete variables may be referenced by qualification of \( a' \), i.e. \( a'.c_0, \ldots, a'.c_{n-1} \). As a matter of convenience within module \( M \), the concrete variables \( \overline{c} \) introduced in the interface of \( M \) to represent \( a \) may be used without this qualification although each \( c \in \overline{c} \) can also be referred to as \( a'.c \).

4.3.1. Summary of identifier scope, use and meaning

In the previous section we noted the various places in an implementation module where identifiers can be introduced and gave each such set of introductions a name. In the interface we introduced (1) the module parameters \( \overline{G} \), (2) the value parameters \( \overline{n} \), (3) the abstract variables \( \overline{a} \), (4) the concrete variables \( \overline{c} \) and (5) the type variables that occur free in the type expression \( \Sigma \). Within the interface, free type variables in \( \Sigma \) can be used in the representation (concrete) type expression \( \Pi \), while the parameters \( \overline{G} \) and \( \overline{n} \) may be used in any implementation directives associated with the concrete variables. The coupling, as mentioned, may reference \( \overline{a} \), \( \overline{c} \) and \( \overline{n} \) and \( I_G \) for any \( G \in \overline{G} \). The identifiers in (1) - (5) are the only identifiers that may occur free throughout a module and should all be distinct.

The identifiers \( h \) are associated with a template or program fragment in the definition section. This is a notational convenience, and the \( h \) can be eliminated by replacing them by their definitions in the remainder of the module.

Within an implementation unit, in addition to the identifiers introduced in the interface, new identifiers \( \overline{\nu} \) can be introduced to represent the (value of) expressions or statements matched by the nonterminals or the value of arguments to modules named in the template. The scope of such an identifier is limited to the implementation unit in which it is introduced. The identifiers \( \overline{\nu'} \) are the representation of the values \( \overline{\nu} \).

The precondition and the template can refer only to abstract variables, while the implementation refers exclusively to representation variables. Therefore the template and implementation do not share any variables.

4.4. Cooperation Between Modules

Different operations are implemented by different modules, yet a complete program must have a single implementation. The module parameters and the implementation restrictions on nonterminals in templates make it possible to write modules that, taken together, provide an implementation for a complete program over several variables. For example, the implementation of set operations by the BITV module in Fig. 3 yields PASCAL code, and this requires considerable cooperation with implementations of other variables since they must yield PASCAL code as well. Indeed all implementations must cooperate with a general implementation of statements since the abstract syntax for our programming notation must be related to PASCAL abstract syntax. It is these requirements that have lead to the proliferation of implementation names in the BITV module. In this section we briefly examine modules that support module BITV in order to provide implementations for complete programs.
module P_INT impl 

impl i : Int ~ i' : Integer

 coupling ~(maxint + 1) ≤ i = i' < maxint

 scope 

   {true} ( i / P_INT || i = 0 || (stmt S) : P_STMT) : P_STMT

   ~ begin var i' : Integer;

   i' := 0;

   S'

   end

 stmt

   {i + 1 < maxint} (i := i + 1): P_STMT ~ i' := i' + 1

 exp

   {true} i : P_INT ~ i'

   {true} (i < (\exp e): P_INT): P_BOOL ~ i' < e'

   {true} (2 | i): P_BOOL ~ 0 = i' mod 2

Fig 5. P_INT implementation module.

To begin, module P_INT implements operations on integers. The P_INT module in Fig. 5 is a very limited implementation of Int values suitable only for an upcoming example. Within it, integers in the range ~(maxint + 1).maxint are represented as PASCAL integer values.

One of the three expressions implemented by P_INT is the value of i: the corresponding PASCAL expression is simply the value of the PASCAL variable i'. The other two expressions implemented yield boolean values. Recall that the suffix : P_BOOL following the template asserts that an expression matched by the template is related to its implementation by the coupling in P_BOOL (rather than P_INT which only couples the first expression and its implementation).

Module P_BOOL is not shown because the operations given in P_INT will suffice for this exposition. The coupling of P_BOOL is simply a = a', since all boolean values can be completely represented; a general implementation of P_BOOL might implement all operations except quantification.

The scope unit in P_INT implements the introduction of an Int variable i as a PASCAL block in which i' is introduced and declared to be a PASCAL Integer. The body of the block is the implementation S' of the statement S.

Remark

Actually, variable declarations are not permitted inside PASCAL begin .. end blocks, so that this is not a translation to true PASCAL. It is not possible to provide a true PASCAL implementation of our scoped blocks because the translation is entirely syntax-directed and the syntactic structure of our notation is fundamentally different in this respect from PASCAL. We could consider more sophisticated forms for the right hand side of an implementation unit that would allow us to build up several parts of the translation (such as declarations and program statements) separately and simultane-


ously. However, as we shall see later, writing implementations to translate to a different notation will be the exception rather than the rule, and if necessary at all, such translations form a last step before compilation and execution. Therefore we might just as well translate to "byte-code" or some other language that we arrange to be a more appropriate target for a syntax-directed translation of our notation. PASCAL is used here for didactic reasons.

Module P_STMT given in Fig. 6 illustrates the implementation of statements; no variables need be represented hence the impl portion and the coupling are omitted. The purpose of the module is to provide a PASCAL implementation of a small subset of statements of the general programming notation. Note that the template for the if statement makes use of a repeated non-terminal label B to match a class of alternative statements that can be implemented with a single-arm conditional.

We shall be making use of these modules in a program following the definition of implementation directives.

4.5. Implementation Directives

A module instantiation \( \hat{M} \) is an expression of the form \( M \ [ \overline{H} \ (\overline{f}) \] \) where \( M \) is a module name, the \( \overline{H} \) are themselves module instantiations agreeing in number with the module parameters in the interface of \( M \), and the \( \overline{f} \) are expressions agreeing in type and number with the value parameters in the interface of \( M \). For example, \( \text{BITV}(n+1) \) is a module instantiation of the BITV module.

An implementation directive \( \overline{a} / \hat{M} \) is specified by combining a module instantiation \( \hat{M} \) with the introduction of zero or more variables \( \overline{a} \). Variables that are to be implemented together must therefore be introduced together. The scope of the directive \( \overline{a} / \hat{M} \) is the scope of the variables \( \overline{a} \). The directive associates this scope with a second, parallel, scope which introduces variables \( \overline{a}' \) representing \( \overline{a} \) and constructs corresponding implementations in this scope for

```plaintext
module P_STMT
defn
  stmt S \equiv \langle stmt S \rangle : P_STMT
  stmt T \equiv \langle stmt T \rangle : P_STMT
  exp B \equiv \langle exp B \rangle : P_BOOL

stmt
  (true) S; T \sim S'; T'
  (true) do B \rightarrow S od \sim \text{while } B' \text{ do } S' \text{ end}
  (true) if B \rightarrow S \sim \text{if } B' \text{ then } S'
       [] \rightarrow B \rightarrow \text{skip}
       fi
```

Fig 6. P_STMT implementation module.
statements and expression matched by the implementation units of \( M \).

Not every variable in a program is introduced in a scope; the variables that occur free within the program have no enclosing scope. An implementation directive for a variable \( a \) that occurs free in a program \( S \) is written as \( a / M \vdash S \), and the scope of the directive is the entire program \( S \). Such an implementation directive is really an assumption about the representation of \( a \) that is to be relied upon in providing implementations for the operations on \( a \) in \( S \). Any claims about the syntactic and semantic correctness of an implementation of a program are conditional on the implementation of its free variables.

At most one implementation directive can apply to a given variable. We will be working under the assumption that each variable in a program is associated with some implementation directive, although that directive may often be provided by default. Consequently, every variable will be associated with exactly one implementation directive and, in a program suitably decorated with implementation directives, all parts of the program and the program itself will have an implementation. The association of an implementation \( W' \) with a program fragment \( W \) under the direction of a set \( A \) of implementation directives will be called a translation and will be written as follows

\[
A \vdash (W \sim W') : N \tag{4.1}
\]

\( N \) is the module that couples the values of \( W \) and \( W' \) and is usually a module in the set of implementation directives \( A \), but not necessarily so.

Some restrictions apply to an implementation directive and its arguments. The implementation directive must be applied to the right type of variables; that is, the inferred type of the program variables must be a substitution instance of the type expression \( \Sigma \) in the module interface.

As concerns the arguments that instantiate an implementation, the number of arguments and their agreement with the parameters in the module interface can easily be checked with the type inference system. As an additional requirement, the instantiation of module \( M \) in the implementation directive \( a / M \) must be constant throughout the scope of the directive. This is certainly the case for any module argument since implementation modules are not defined within a program. A value argument, on the other hand, can be an expression \( f \) in program variables that change within the scope of the directive. If \( S \) is the statement portion of the scope, we can insure that \( f \) has the same value anywhere in \( S \) by requiring that (1) \( \text{fv}.f \cap \text{def} \ S = \emptyset \) (see section 2.3.2) and (2) any scopes contained in \( S \) have their bound variables relabeled to avoid capture of \( \text{fv}.f \). The reason we require the module instantiation to be constant in the scope is that the instantiated parameters can appear in module couplings which must be invariant throughout the scope.

4.5.1. Translating programs with implementation directives

Our goal is to place implementation directives in a program in such a way that the entire program can be translated. We examine the translation process for a sample program annotated with implementation directives specifying the use of the implementation modules defined in section 4.4.
To place this process properly in the context of program development, suppose we set out to develop a program to find the set $a$ of nonnegative even integers smaller than $n$. To be precise, we might start with the specification

\[(0 \leq n, a = \{i \mid 0 \leq i < n \land 2 \mid i \})\]  \hspace{1cm} (4.2)

There are many ways to refine such a specification, although the principal stumbling block in a standard derivation would be that $a$ denotes a set rather than a more efficient array. In our approach we take advantage of the fact that set $a$ can be represented efficiently using the BITV implementation for $a$. Consequently we refine (4.2) to a point where all expressions and statements involving $a$ are within the restricted collection supported by module BITV. An example of program development "aimed" at using certain operations is presented in [Gries86]. Without going into detail about such steps here, we might arrive at the following refinement $S_{\text{prog}}$ satisfying (4.2):

\[S_{\text{prog}} \equiv a := \{ \}; \hspace{1cm} (4.3)\]

\[(i \mid i = 0 \mid \text{do } i < n \rightarrow \text{if } 2 \mid i \rightarrow a := a \cup \{i\} \]
\[\{k \rightarrow \text{skip} \]
\[\text{fi} ; \]
\[i := i+1 \]
\[\text{od} \]

The main point we have been stressing about such a derivation is that it does not involve a change of notation: at all times $a$ is a set with the operations and identities shown in Fig. 12 in the Appendix. At any point in the derivation we can attach a set of implementation directives to the variables in (4.3), and investigate the syntactic suitability of the modules directed for use thereby. If the implementations are suitable then a translation of $S_{\text{prog}}$ is provided by the modules.

**Selection of Implementations**

We are not concerned at the moment with the inference of implementation directives—even if a syntactically suitable set of implementation directives could be found, the semantic issue of their adequacy still needs to be addressed—so we attach "trial" implementation directives to the variables of $S_{\text{prog}}$. We choose to introduce $i$ as $i / \text{P\_INT}$ in $S_{\text{prog}}$ and to assume $a$ is implemented with BITV($n$) and $n$ is implemented with P\_INT. The representation of $n$ as PASCAL integer $n'$ is required in order to implement the guard in the iterative statement, but also because $n$ is constrained to be implemented by P\_INT in the interface of BITV. We also assume the implementation P\_STMT (of no variables) to translate from our notation to PASCAL. If $S_{\text{prog}}^*$ denotes (4.3) with the implementation directive P\_INT attached to the introduction of $i$, our full annotation is written

\[\text{P\_STMT, } n / \text{P\_INT, } a / \text{BITV}(n) \vdash S_{\text{prog}}^*\]

**Translation**

Some fragments of $S_{\text{prog}}$ (4.3) can be translated directly by a single module. For example, module P\_INT given in Fig. 5 provides the translation
\[ i / P \_INT \vdash (21i \sim 0 = i' \mod 2) : P \_BOOL \]  

(4.4)

Other fragments can be translated only if the same or different modules are able to translate components of the fragment. For example, we have

\[ a / BITV(n) \vdash (a := a \cup \{ i \} \sim c(i') := true) : P \_STMT \]

only if some module can provide \((i \sim i') : P \_INT\), since this is what the template in BITV calls for in Fig. 3. Module \(P \_INT\) is shown in Fig. 5 and does indeed provide such a translation if \(i\) is introduced with implementation directive \(P \_INT\); i.e. \(i / P \_INT \vdash (i \sim i') : P \_INT\). Combining these requirements we find that

\[ a / BITV(n), i / P \_INT \vdash (a := a \cup \{ i \} \sim c(i') := true) : P \_STMT \]

(4.5)

provides the complete translation for the fragment.

Similarly, module \(P \_STMT\) can translate the two arm if guarded command if the top arm can be translated as a \(P \_STMT\), and the top guard can be translated as a \(P \_BOOL\). These are precisely the translations developed in (4.5) and (4.4), respectively, so that we obtain

\[
P \_STMT, a / BITV(n), i / P \_INT \vdash 
\begin{align*}
&\text{if } 21i \to a := a \cup \{ i \} \sim \text{if } 0 = i' \mod 2 \text{ then } c(i') := true \\
&\text{fi }
\end{align*}
\]

The requirements to the left of the turnstile fix the implementations of free occurrences of variables in the fragment; a requirement such as \(i / P \_INT\) can be discharged by an implementation of a scope in which \(i\) is introduced with implementation directive \(i / P \_INT\). We can continue in this fashion to form translations of ever larger fragments of \(S^*_{prog}\) until we obtain the translation

\[
P \_STMT, n / P \_INT, a / BITV(n) \vdash (S^*_{prog} \sim S'_{prog}) : P \_STMT
\]

(4.6)

where

\[
S'_{prog} \triangleq \text{for } j := 0 \text{ to } n' - 1 \text{ do } c(j) := \text{false}; \\
\begin{align*}
\begin{array}{l}
\text{begin} \\
\text{var } i' : \text{Integer}; \\
i' := 0; \\
\text{while } i' < n' \text{ do} \\
\text{if } 0 = i' \mod 2 \text{ then} \\
c(i') := \text{true}; \\
i' := i' + 1;
\end{array}
\end{align*}
\]

The hypotheses in (4.6) are exactly the implementation directives we chose for the free variables of \(S^*_{prog}\), and this assures us that the implementation directives chosen are syntactically suitable. The implementation itself is the right hand side \(S'_{prog}\) of a translation of \(S^*_{prog}\).

4.5.2. Syntactic suitability of implementation directives in a program

The process whereby we build up the translation of an ever larger program fragment using assumptions about the implementations of the free variables in the fragment can be described by a formal system. A formal system that characterizes the valid translations can be used to
determine whether a program is annotated with syntactically suitable implementation directives as well as the translation itself.

An implementation directive $\tilde{a} / M \{ \tilde{H} \} \{ \tilde{f} \}$ adapts the module $M$ by instantiation of the module interface. This is most conveniently viewed as syntactic substitution into the implementation units of $M$. The module parameters are replaced by their instances $\tilde{H}$, the value parameters are replaced by their instances $\tilde{n}$, and the abstract variables are replaced by $\tilde{a}$. Each implementation unit adapted in this fashion can be presented as an inference rule in the formal system to derive valid translation formulae of the form (4.1).

Within an implementation unit there are two ways to require values to have a certain implementation. Let $e_0, \ldots, e_k$ be the labels of the nonterminals in the template, and let $E_0, \ldots, E_k$ be the (adapted) implementation constraints attached to the nonterminals. Let $f_0, \ldots, f_l$ be the arguments to the module that are used in the template and let $F_0, \ldots, F_l$ be the (adapted) constraints attached to those arguments in the interface. Let $T(\tilde{e}, \tilde{f})$ be the template with the nonterminals and parameters to the module replaced with the variables $\tilde{e} = e_0, \ldots, e_k$ and $\tilde{f} = f_0, \ldots, f_l$, which will be variables of the inference rule. Let $W(\tilde{e}', \tilde{f}')$ be the implementation that refers to the corresponding representations of the parameters $\tilde{e}$ and $\tilde{f}$. Finally, let $N$ be the module coupling $T(\tilde{e}, \tilde{f})$ with $W(\tilde{e}', \tilde{f}')$, i.e. the implementation unit has the form $T : N \vdash W$. The inference rule is

$$
A, a / M \{ \tilde{H} \} \{ \tilde{f} \} \vdash (e_0 - e'_0) : E_0, \ldots, (e_k - e'_k) : E_k, \\
A, a / M \{ \tilde{H} \} \{ \tilde{f} \} \vdash (f_0 - f'_0) : F_0, \ldots, (f_l - f'_l) : F_l
$$

(4.7)

where $A$ is any set of implementation directives not involving $a$. If $T$ is a scope incorporating the implementation directive, the directive does not appear in the consequent. If the template contains no constrained non-terminals and does not depend on the implementation of the module arguments, then the inference rule is an axiom schema.

To complete the formal system we need an inference rule to import more implementation directives in the translation hypothesis, provided they do not involve variables already implemented.

$$
A \vdash (S \vdash S') : N \\
A, a / M \vdash (S \vdash S') : N
$$

provided $a \notin A$

If $S$ is completely decorated with implementation directives, and $B$ is a set of implementation directives for the free variables of $S$, then the implementation directives are syntactically suitable if it is possible to derive a formula

$$
B \vdash (S \vdash S')
$$

where $S'$ is the implementation of $S$.

### 4.6. Towards More Flexible Modules

The BITV implementation discussed, in cooperation with other modules, allows us to translate certain programs over integer sets into PASCAL programs. But the BITV module, as given, is not very flexible. For example, it only provides a bitvector implementation when one certain implementation of integers (P_INT) represents the values being inserted in the set. It
requires an implementation of P_BOOL and P_STMT to support the translation to PASCAL. If
we are to build libraries of modules we shall need them to be as flexible as possible in their
applicability, so that they may be used with many clients. We sketch three techniques for
improving the flexibility of modules.

4.6.1. Subset translation

Perhaps the reader has noticed the "mismatch" between the notation on the left and right
hand side of module BITV in Fig. 3. The type of the concrete variable
(array [0..n'-1] of Integer) has nothing to do with the type system for our notation. This is a
sign of a larger problem that becomes more troublesome when we want to establish the correct-
ness of an implementation. Since the PASCAL notation differs from ours, we don’t have a
meaning for the statements and expressions on the right hand side of a module. In the coupling
we treated the PASCAL variables as if they were in our notation, and, in fact, their meaning is
pretty close and can certainly be modeled with our notation.

In general, it is far more convenient to keep the translation entirely within our notation. If
the abstract and concrete notations are the same, then the type agreement and correctness are far
simpler to establish, and the need for implementations like P_STMT, which make systematic
changes to the abstract syntax, is eliminated. In fact, the "default" implementation for a vari-
able of type \( \tau \) can translate all operations of type \( \tau \) to themselves, effectively leaving all opera-
tions on the variable intact.

But since these translated programs are still in our notation, how can they be executed?
PASCAL notation, after all, has the advantage that it can be compiled to produce an executable
program. Our approach is to select some restrictions of our notation that correspond directly to
efficient operations on data structures natural to a target computing engine. We can describe
such modules without the low-level details of their implementation by presenting them as "sig-
natures", consisting solely of the abstract portion of the interface and implementation units.

A signature for the implementation module INT_ARRAY is given in Fig. 7. This module
makes a significant restriction of the sequence notation that is efficiently implemented as an
integer array, so clients of INT_ARRAY will have to refine their programs considerably. Their
sequence \( b \) must be introduced with a domain predicate \( \#b = k \), giving \( b \) some initial length \( k \)

```
module INT_ARRAY (n) impl b : Seq(Int)
  coupling \( \forall (i \mid 0 \leq i < n) \ll n \ll \ll (b_i, b'_i) \)
  scope
  \{ k \leq n \} \ll (b i \ll\ll\ll (exp_k) \ll \ll (stmt S))
  exp
  \{ true \} \ll b\langle exp_i \rangle : INT
  stmt
  \{ true \} \ll b := (b; \langle exp i \rangle : INT) (\langle \exp e \rangle : INT)
```

Fig 7. INT_ARRAY implementation signature.
(which must be less than \( n \) —see the unit precondition), but without any specific values. Since the only implemented statement does not change the size of \( b \), \#b is constant throughout the scope of \( b \) in any syntactically suitable client of \( \text{INT\_ARRAY} \). Consequently the module need not include unit preconditions to ensure that index \( i \) satisfies \( 0 \leq i < n \), because this follows directly from the correctness of its client.

To take advantage of the executable implementation of \( \text{INT\_ARRAY} \) and other “provided” modules, client programs would have to be refined so that all variables are implemented using only these modules. This is, in itself, unsatisfactory. If the efficient notational restrictions provided are, for example, finite integers and fixed size sequences, then a specification involving sets will require changes to the specification to rephrase it in terms of sequences. This is exactly the sort of change we are trying to avoid in program development, by encapsulating it in implementation modules.

The solution is to allow the use of efficient implementations to be directed on the right hand side of implementation modules. That is, if \( \text{BIT\_ARRAY}(k) \) is one of the efficient restrictions of \( \text{Seq}(\text{Bool}) \), then a module like \( \text{BITV} \) couples \( a : \text{Set}(\text{Int}) \) to \( c : \text{Seq}(\text{Bool})/\text{BitArray}(k) \). A translation using this form of the \( \text{BITV} \) module will yield a program containing implementation directives. The directives might be treated as pragmas to a compiler that operates on our notation. Such a compiler might refuse to compile programs with implementation directives outside of the efficient set.

Alternatively the translated program can be translated again, under control of the inserted implementation directives. The second translation would involve a change in notation, in order to end up with an executable form. As we mentioned, the development and correctness of a module that changes notation is more complicated. But we do not believe that there are fundamental problems developing and proving very low-level implementations; the primary difficulty is the definition of an appropriate semantics (low-level notations have cumbersome semantics) and carrying out the very detailed module-correctness proofs generated as a result of complex semantics. The correctness of couplings between high-level operations and low-level implementations (machine code for a Floating Point Systems Array Processor) is considered in [Prins86].

In either case, programs can be developed over a variety of data types, with general implementation modules coupling various restrictions of the data types to a smaller set of types for which efficient implementations can be directed on the right hand side of the modules. The correctness of both the programs and the modules is considered entirely within our defined notation.

### 4.6.2. Generic implementations

Module \( \text{INT\_ARRAY} \) implements only one kind of sequence: a sequence of integers whose values are represented by module \( \text{INT} \). If we happened to have a different implementation for integers (say halfword representations, or perhaps arbitrary range integers), we would need a different \( \text{INT\_ARRAY} \) module to provide implementations for arrays of such values. And, more generally, it would be preferable not to have to build classes of different ARRAY implementations for different types.
module ARRAY [M] (n) impl b : Seq(σ)
coupling ∀(i : 0 ≤ i < n ∀ lM(b, b'_i))
scope {k ≤ n} (b / ARRAY [M] (n) || #b = (exp k) || (stmt S))
exp {true} (b (exp i) : INT) : M
stmt {true} b := (b; ((exp i) : INT) ; ((exp e) : M))

Fig 8. Generic ARRAY implementation signature.

The module interface allows us a far more general definition of a module to implement sequences of any type, using an implementation valued parameter M to stand for the implementation of the element type. The generic definition of ARRAY is given in Fig. 8.

Generic modules commit us to a compiler pragma view of the implementation directives in the translated program. Generic implementations will be fully instantiated in the translated program, so that although the translated program will always specify fixed implementation directives, the possible instances may be unbounded. Suppose, for example, the implementation directives "recognized" by our compiler are INTFW, INTHW, BOOL and ARRAY [M] (n). The compiler must be prepared to implement the set of operations in Fig. 8 for each particular instantiation for M: ARRAY [INTFW] (n) and ARRAY [BOOL] (n) and ARRAY [ARRAY [INTFW] (n)] (n) and so forth.

Generic modules make greater demands on the interpretation of translated programs, with a potential decrease in efficiency. Yet they are very attractive because they are extremely reusable. We see a strong role for generic modules in the rapid prototyping of programs; less efficient generic implementations can be used at an earlier stage of program development.

4.6.3. Coercions

Even in the generic form of (Fig. 8) ARRAY depends on the INT implementation for index i. Can the implementation of integers as index into the array not be made a parameter too? To do so could actually reduce the flexibility of the ARRAY module. Consider that the implementation of array indexing must be able to interpret the representation of the index i. Suppose we built some unusual implementation of integers particularly suited to the way our program uses integers. The implementation directives recognized by our compiler would not include our implementation; the compiler can not be expected to determine from our coupling how an integer can be recovered from its representation in our module. Nevertheless, we would like to be able to use our implementation of integers in conjunction with other implementation modules that expect "standard" integers, say implemented by INT. The way we can arrange this is to provide "coercion" between our implementation and some other, more general, implementation. That is, for our implementation of integers we provide a way to materialize a standard INT value and vice versa.
module TWO \impl \ i : \Int \sim v, b : \Int \times \Bool \\
coupling \quad i = v \land b \equiv 2 \mid v \\
scope \\
\{ true \} \quad (i \app TWO \app i = 0 \app \langle \text{stmt } S \rangle) \\
\quad \sim (b \app BOOLEAN \app b \app (v \app \Int \app v = 0 \app \langle S' \rangle)) \\
stmt \\
\{ i < \maxint \} \quad i := i + 1 \quad \sim \quad v := v + 1; b := \neg b \\
exp \\
\{ true \} \quad 2\,i : BOOLEAN \quad \sim \quad b \\
\{ true \} \quad \langle\langle \exp e \rangle : TWO \rangle : \Int \quad \sim \quad e' . v \\
\{ true \} \quad \langle\langle \exp e \rangle : \Int \rangle : TWO \quad \sim \quad (e', 2 \mid e') \\

Fig 9. Implementation TWO with coercion to and from INT

Consider the implementation TWO given in Fig. 9. It represents an integer \( i \) as an integer \( v \) and a truth value \( b \), the latter recording whether \( i \) is even. Our intention is to provide a faster implementation of \( 2 \mid i \) in a program where \( i \) is subject to change by the statement \( i := i + 1 \). But we do not want to have to provide implementations for all the ways \( i \) might be used as an integer. Consequently we provide coercion to and from the more general implementation \( \Int \) of integers with the the last two expressions implemented in module TWO. Module TWO can be used directly as a better implementation directive for \( i \) in refinement (4.3). Including coercions allows us to connect a specialized module to an existing collection of more general modules conveniently and flexibly.

If we connect each new implementation of integers to some other, already constructed, implementation by coercion to and from the established implementation, then, in principle, we could convert an integer from any representation to any other representation by a sequence of coercions. Of course, coercions from a specialized to a more generalized implementation will have a precondition that is not identically \textit{true} in at least one of the directions, so that at best only some values will be arbitrarily interconvertible. The precondition for coercion to and from \Int in module TWO is \textit{true} because each integer represented by TWO can be represented as an \Int and vice-versa.

The inclusion of coercions in implementation modules leads to some new considerations in the determination of syntactic suitability of implementation directives. First, since coercions may yield results coupled by a module different from the set of modules specified with implementation directives, the formal system must be extended by rules contributed by modules mentioned in coercions. Second, the formal system no longer has the property that each rule of inference increases the size of the abstract program fragment in the translation formula. Care must be taken not to generate translations that perform redundant coercions such as TWO to \Int back to TWO back to \Int.

The three mechanisms presented—implementation directives on the right hand side of modules, module valued parameters and coercions—allow us to build more generally applicable
modules more conveniently. The next and last section of this chapter considers the development of an implementation module.

4.7. Implementation Development: HEAP

We illustrate the construction of an implementation module with a simple example. The implementation is based on the observation that a set $a$ whose values are totally ordered under a relation induced by a function $f$ (the $\text{min}$ function, for example, on a set of integers), can be represented in a sequence $c$ as a heap that keeps values ordered according to $f$ (see, for example, [Aho74]). Such a representation admits insertion of new elements or the deletion of the "minimum" element $f/a$ using $O(\log \#a)$ applications of $f$ and gives constant time access to $f/a$.

**Interface.**

From the description above we can directly write the interface for the module as

```plaintext
module HEAP (f) impl a : Set(\sigma) \rightarrow c : Seq(\sigma)
```

We have not given the type of parameter $f$ since it can be inferred from operations that we will be implementing. The appropriate type expression is $f : \sigma \times \sigma \rightarrow \sigma$.

Nothing in the interface definition fixes the type of elements in set $a$, and since $\sigma$ appears in the type of $c$ this definition is generic. In the initial development of a module, the representation of the concrete variables and the implementation of nonterminals in templates is not an issue; this concern is factored in later on, exactly as in the development of programs. The initial implementation module we construct here will simply translate some set operations in our notation to some sequence operations in our notation.

**Coupling.**

With our interface defined, we can formalize our intuition about the representation in the form of a coupling predicate. Each element of $a$ should appear exactly once in sequence $c$, so $a = [c] \land \#a = \#c$. A binary tree is embedded within the sequence $c$ in such a way that the immediate ancestor of $c_i$ is $c_{(i-1) \div 2}$, and the ancestor relation in the tree respects $f$. This gives us

$$\text{coupling} \quad a = [c] \land \#a = \#c \land \forall (i \parallel 0 < i < \#c \parallel c_{(i-1) \div 2} = f(c_i, c_{(i-1) \div 2}))$$

The only variables that occur free in $l$ are $a$, $c$ and $f$, all introduced in the interface.

**Implementation Units.**

We undertake now an investigation of terms involving $a$, identifying those that we can easily implement. For example, the coupling tells us that the "smallest" element (according to $f$) in $a$ is at $c_0$; hence we can implement $f/a$ by $c_0$:

```
exp {true} f/a \sim c_0
```

The precondition expresses our confidence that whenever $f/a$ is defined, then $c_0$ is also and their values are identical. Similarly, by direct inspection of the coupling we find that the size of $a$ and $c$ are the same, so

```
exp {true} \#a \sim \#c
```
Next we consider statements. Our goal is to *preserve* the coupling between \( a \) and \( c \) under the combined execution of an abstract operation and its implementation. Suppose \( a \) is assigned the value \( \emptyset \); the assignment that establishes (4.8) is \( c := \emptyset \), so we write

\[
\text{stmt } \{ \text{true} \} \quad a := \emptyset \quad \sim \quad c := \emptyset
\]

A more difficult operation is the update of \( a \) with a new element \( v \), which we characterize with the template \( a := a \cup \langle \langle \text{exp} \ v \rangle \rangle \). Clearly \( v \) must be placed in \( c \) to preserve the coupling. But any additional to \( c \) must also preserve the heap-property. If we append \( v \) to \( c \) it will be a leaf of the binary tree, and to restore the heap property we repeatedly exchange the value of \( v \) with that of its ancestor until the ordering is re-established or \( v \) arrives at the root. The following loop accomplishes this.

\[
\text{defn stmt bubble_up \equiv}
\begin{align*}
(i \equiv \text{true} \quad & i := \#c - 1; \\
\text{inv } & \forall (j \equiv 0 < j < \#c \land j \neq i \equiv c_{(j-1) \div 2} = f(c_j, c_{(j-1) \div 2})) \\
\text{do } & i > 0 \land c_{(i-1) \div 2} \neq f(c_i, c_{(i-1) \div 2}) \rightarrow \\
& c_i, c_{(i-1) \div 2}, i := c_{(i-1) \div 2}, c_i, (i-1) \div 2 \\
\text{od}
\end{align*}
\]

With the insertion of \( v \) in \( a \) we encounter situations where the abstract operation is defined, but the implementation could fail. Since \( a \) is a *set*, it will not change if \( v \) is already a member of \( a \), so that insertion of \( v \) in \( c \) could violate (4.8). From our coupling we see no convenient way to determine whether \( v \in a \), so we eliminate that possibility by adding \( v \notin a \) to our precondition. The second point is more subtle: the heap property depends on the totality of the ordering induced by \( f \) on \( a \cup \{ v \} \). We can guarantee that \( f \) induces a total ordering if

\[
\forall (x, y, z \equiv x, y, z \in a \cup v \equiv f(x, y) \in \{ x, y \} \\
\phantom{\forall (x, y, z \equiv x, y, z \in a \cup v \equiv f(x, y) \in \{ x, y \} \quad \wedge} f(x, y) = f(y, x) \\
\phantom{\forall (x, y, z \equiv x, y, z \in a \cup v \equiv f(x, y) \in \{ x, y \} \quad \wedge} f(x, f(y, z)) = f(f(x, y), z)
\]

This is not a simple requirement to include as part of the unit precondition; its repeated verification within a client program would be tedious. We can eliminate (4.9) from the precondition by including it in the coupling. Of course, pushing extra requirements into the coupling will make it harder to establish it initially, but since this is only done once, it is, in general, advantageous. But as it stands, (4.9) references \( v \), which is precluded inside a coupling. We eliminate \( v \) by generalizing the domain predicate of (4.9) to \( \text{true} \), so that it holds for all possible \( a \) and \( v \),

\[
P_f: \forall (x, y, z : \sigma \equiv \text{true} \equiv f(x, y) \in \{ x, y \} \land f(x, y) = f(y, x) \land f(x, f(y, z)) = f(f(x, y), z)
\]

and include it in the coupling (4.8). This gives us the implementation unit

\[
\text{stmt } \{ v \notin a \} \quad a := a \cup \langle \langle \text{exp} \ v \rangle \rangle \\
\phantom{\text{stmt } \{ v \notin a \} \quad a := a \cup \langle \langle \text{exp} \ v \rangle \rangle} \quad \sim \quad c := c ^ "[v]"; \text{bubble_up}
\]

The template does not constrain \( v \) to have a certain implementation; those considerations are factored in later. Without a restriction on the implementation of expression \( v \), the coupling of \( v \)
to its representation is simply $v = v'$.

Remark

The relation $\leq_f : \sigma \times \sigma \rightarrow \text{Bool}$ is induced by $f$ with the definition

$$x \leq_f y \iff x = f(x, y)$$

To see that $\leq_f$ defines a total ordering, observe that

[reflexivity]

$x \leq_f x$ means $x = f(x, x)$ which is guaranteed by the first clause of (4.10)

[anti-symmetry]

$x \leq_f y \land y \leq_f x$ means $x = f(x, y) \land y = f(y, x)$ and using commutativity in (4.10) we get

$x = f(x, y) = f(y, x)$ therefore $x = y$.

[transitivity]

$x \leq_f y \land y \leq_f z$ means $x = f(x, y) \land y = f(y, z)$. By substitution we obtain $x = f(x, f(y, z))$

which is $x = f(f(x, y), z)$ by associativity in (4.10). Another substitution yields $x = f(x, z)$ which means $x \leq_f z$

[totality]

$x \leq_f y \lor y \leq_f x$ means $x = f(x, y) \lor y = f(x, y)$ which is implied directly by the first clause of (4.10).

We chose the restrictions in (4.10) on $f$ because the commutativity and associativity of $f$

follow immediately from the correctness of any client program that includes the expression $f / a$. Note that we cannot assume the client of a module uses all operations of the module, so we can not assume $f$ will be commutative and associative in all clients. Since we rely on this fact in the arrangement of elements in $c$, it must be included as a proof obligation. In general the only assumption we can make is that a valid client of an implementation module $M$ of a variable $a$ uses only operations in $M$ on $a$.

The next statement we investigate involves deletion of elements from $a$. From the coupling we see that the only element whose location in the heap is known is the minimum element in $a$; locating other elements requires a search of some sort. Therefore we restrict the statements deleting elements of $a$ to the single statement $a := a - \{f / a\}$, which deletes the smallest element in $a$. To implement it, we must eliminate $c_0$, which means we must find a new root for the heap. A simple scheme is to replace $c_0$ with the last element in $c$, and then repeatedly exchange this value with its smaller descendant until the ordering is re-established. This requires a simple loop $\text{bubble\_dn}$, similar to $\text{bubble\_up}$ above, which is not elaborated here. The implementation maintains the coupling without any further assumptions, hence we have

$$\text{stmt} \quad \{ \text{true} \} \quad a := a - \{f / a\}$$

$$\sim \ c := \lfloor c(\#c - 1) \rfloor \ c(1..\#c - 2); \text{bubble\_dn}$$

Which operations we choose to implement with a module is entirely up to us to decide. If we pick a very small collection, then the module will not be very useful. A very large collection will be tedious to implement and might lead to inefficient operations that are best implemented using another representation. For example, we have already observed that it is difficult to determine membership of a given value $v$ in $a$ using the HEAP representation. It is nevertheless possible to give an implementation for the expression $v \in a$, for example by sequential
search through \( c \). But it would probably be more appropriate to build a different module using another representation, for example an AVL-tree, to implement all the operations implemented by HEAP along with fast element lookup and arbitrary element deletion. An AVL implementation will probably be more expensive in time and space on the operations it has in common with the more restricted HEAP module (such as \( f / a \)).

There are a few more operations that fit nicely with the heap representation strategy, and illustrate some points. Nondeterministic statements, such as \( \text{choose } v \text{ so } v \in a \), leave a great deal of freedom in their implementation. A (client) program that incorporates such a statement is correct for any choice of \( v \), so that an implementation is free to select a particular choice for \( v \). Although it was not efficient to delete a given element from \( a \), we can give a very efficient implementation of the extraction and deletion of an arbitrary element from \( a \), since removal of the rightmost leaf of the heap (the last element in \( c \)) does not alter the heap property.

\[
\text{stmt} \{ \text{true} \} \; \text{choose} \; (id \; v) \; \text{so} \; (id \; v) \; \in \; a; \; a := a - \{(id \; v)\} \\
- v', \; c := c(\#c-1), \; c(0..\#c-2)
\]

Another case where non-determinism in the client leaves the opportunity for an efficient implementation is a statement that iterates over the elements of \( a \). Since \( a \) is a set, no order is implied by iteration, so we are free to choose an order that suits the representation. Here is an implementation unit that employs a linear traversal of \( c \):

\[
\text{stmt} \{ \text{true} \} \; \text{foreach} \; (id \; v) \; \text{in} \; a \; \{ \text{stmt} \; S \} \\
- \text{for} (v' \; \text{in} \; c \; \{ \text{stmt} \; S \})
\]

The last unit we consider is the implementation of the scope of \( a \). With a scope implementation unit, our job is to establish the coupling for some state satisfying the domain predicate of \( a \). The domain predicate might be very weak, such as \( \text{true} \), or it might make very specific constraints on \( a \), such as \( a = \{ \} \). The weaker the domain predicate, the more freedom left to our implementation. The weakest predicate is \( \text{true} \) and is a good choice in a scope to be implemented since we can rewrite any other domain predicate to \( \text{true} \) by identity (2.1) for blocks. A scope introducing \( a \) satisfying \( \text{true} \) can be implemented as a scope introducing \( c \) satisfying \( c = \{ \} \), since there exists some value for \( a \) satisfying \( \text{true} \) (namely \( \emptyset \) that establishes coupling (4.8) whenever \( c = \{ \} \).

Recall that we added \( P_f \) (4.10) to the coupling (4.8), but that we have no way to establish its truth (since we are implementing \( a \) and not \( f \)). Consequently we must include \( P_f \) as a precondition to the scope implementation unit.

\[
\text{scope} \{P_f\} \; (a \; || \; \text{true} \; || \; \{\text{stmt} \; S\}) \; - \; (c \; || \; c = \{ \} \; || \; S)
\]

We now have all the parts to put together the module HEAP shown in Fig 10, although this module merely translates operations on sets into operations on sequences. Without knowledge of the implementation of the sequence operations, the reader may be unconvinced that the translation improves efficiency. The next stage in the refinement will be to make some representation decisions and to refine the implementation side of the HEAP module.

### 4.8. Refining the HEAP Module

We have given a very general implementation of sets, one that is generic (since the type of \( a \) is not fixed) and one that is independent of the size of the set \( a \) (since it blithely extends the sequence \( c \) to match it). In order to use an efficient implementation for sequence \( c \) we will
module \( \text{HEAP}(f) \) impl \( a : \text{Set}(\sigma) \sim c : \text{Seq}(\sigma) \)

**coupling** \( a = \{ c \} \land \#a = \#c \land P_f \land \forall (i \mid 1 \leq i < \#c \mid c(i - 1) \div 2 = f(c_i, c(i - 1) \div 2)) \)

**defn**
- \( id \ v \equiv \langle id \ v \rangle \)
- \( stmt \ S \equiv \langle stmt \ S \rangle \)
- \( stmt \ \text{bubble}\_\uparrow \equiv \cdots \)
- \( stmt \ \text{bubble}\_\downarrow \equiv \cdots \)

**scope**
- \( \{ P_f \land 0 \leq n \} (a \parallel true \parallel S) \sim (c \parallel c = [\cdot] \parallel S') \)

**exp**
- \( \{ true \} \ f/a \sim c_0 \)
- \( \{ true \} \ #a \sim \#c \)

**stmt**
- \( \{ true \} \ a := \{ \} \sim c := [\cdot] \)
- \( \{ e \in a \} \ a := a \cup \langle \langle \text{exp} \ e \rangle \rangle \sim c := \langle \langle c' \rangle \rangle; \)
- \( \text{bubble}\_\uparrow \)
- \( \{ true \} \ a := a - \{ f/a \} \sim c := [\langle c(\#c - 1) \rangle] \langle c(1..\#c - 2) \rangle; \)
- \( \text{bubble}\_\downarrow \)
- \( \{ true \} \ \text{choose} \ v \text{ so } v \in a; \sim v', c := c(\#c - 1), c(0..\#c - 2) \)
- \( a := a - \{ v \} \)
- \( \{ true \} \ \text{for}(v \parallel v \in a \parallel S) \sim \text{for}(v' \parallel v' \in c \parallel S') \)

**Fig 10.** HEAP module without representation decisions.

Restrict our use of \( c \) in HEAP to a sequence of fixed size \( n \) implemented using the ARRAY implementation of Fig. 8.

We need to make a few alterations to our HEAP implementation in order to use ARRAY. First, in the scope unit of HEAP, the representation of \( c \) as an array is indicated with the directive \( c / \text{ARRAY}[M](n) \). Since ARRAY is generic, we can make HEAP generic by simply adding the implementation valued parameter \( M \) to the HEAP interface. The parameter should be instantiated by the representation of elements in the set. The argument \( n \) of the directive is the maximum size that \( a \) can achieve and should also be a parameter of HEAP. Since \( c \) will now only be used as a fixed size sequence, the number of elements "used" in \( c \) is recorded in a new concrete variable \( k \). The interface has now become

**module** \( \text{HEAP}[M](f, n) \) impl \( a : \text{Set}(\sigma) \sim c : \text{Seq}(\sigma) \)

\( k : \text{Int} \)

If we did not have a generic implementation of ARRAY, then we might further restrict the type designation of \( a \) to, for example, \( \text{Set}(\text{Int}) \). This specializes the typing of \( c \) to \( \text{Seq}(\text{Int}) \) for which there surely is a simple array implementation. Each such specialization reduces the...
domain of applicability of the module, and it is important to build at least some implementations that are generic and that make minimal restrictions. On the other hand, very specialized modules can be very efficient and at least extend the corpus of available implementations for others to use, even though they may not be applicable very often.

The restriction in the size of $c$ means we can only represent certain values of $a$, namely those that satisfy $\#a \leq n$. We strengthen the coupling to reflect the use of $k$ and to limit the size of $a$ with

$$\text{coupling } a = \{c(0,k-1)\} \land \#a = k \leq n = \#c \land P_f$$

$$\land \forall(i \parallel 0 < i < k \parallel c_{(i-1) \div 2} = f(c_i, c_{(i-1) \div 2}))$$

(4.11)

The strengthened coupling and the implementation directive for $c$ lead to some changes in the implementation module we are developing. For example, if $\#a = n$ then $a \cup \{v\}$ may be too large to represent in $c$, so we need $\#a < n$ as a precondition to $a := a \cup \{v\}$. This is the only "externally" visible difference in HEAP: the restriction of an implemented operation to a smaller domain.

Within HEAP, the directive ARRAY $[M](n)$ identifies the collection of operations on $c$ as a client of ARRAY. Since the operations on $c$ as given are not restricted to the set of terms implemented by ARRAY, they need to be rewritten. The refinement of an implementation can be treated just like the refinement of any other program text. Each implementation unit has precondition $I \land \text{preA} \land P$, where $I$ is the coupling, $\text{preA}$ is the set of states in which the corresponding abstract operation is guaranteed to terminate (since only correct uses of the operations in HEAP are assumed to be made), and $P$ is the unit precondition. The coupling is true for a scope implementation unit. For example, the implementation $(c \parallel c = [\ ] \parallel S)$ has precondition $P_f$ in Fig. 10. To restate this unit in a form implemented by ARRAY, incorporating $k$ to establish the coupling we replace it with

$$\text{scope } \{P_f \land 0 \leq n\} (a / \text{HEAP}[M](f,n) \parallel \text{true} \parallel S)$$

$$\sim (c / \text{ARRAY}[M](n) \parallel n = \#c \parallel (k / \text{INT} \parallel k = 0 \parallel S'))$$

The clause $0 \leq n$ is necessary to establish the coupling (4.11), since we don't know anything about $n$.

With the inclusion of implementation directives on the right hand side of the module, we have to show syntactic suitability of those directives in the implementations and we have to prove that the preconditions of the ARRAY implementation are met. All the operations on $c$ in Fig. 10 are implemented in the ARRAY module in Fig. 8, and the interacting identifiers $k$, $v'$ and $e'$ all meet the appropriate implementation constraints (INT, M and M respectively). The precondition for all operations except the scope in ARRAY are true, so the only condition to verify is that

$$P_f \land 0 \leq n \Rightarrow n \leq n$$

which follows immediately.

The final form of the HEAP module is given in Fig. 10. The sample development of HEAP and its subsequent refinement towards a better implementation concludes this chapter on the mechanics of implementation modules and directives. The next chapter discusses the correctness rules for implementation modules and directives that can be used with the formal development of programs and modules.
module \( \text{HEAP}[M](f, n) \) impl \( a : \text{Set}(\mathcal{G}) \) ~ \( c : \text{Seq}(\mathcal{G}) \)
\( k : \text{Int} \)

coupling
\( a = \{ c \} \land \#a = k \leq n = \#c \land P_f \land \forall (i \parallel 1 \leq i < \#c \parallel c_{(i-1) \div 2} = f(c_i, c_{(i-1) \div 2})) \)

defn
\( id \, v \triangleq \langle id \, v \rangle : M \)
\( stmt \, S \triangleq \langle stmt \, S \rangle \)
\( stmt \, \text{bubble}_\text{up} \triangleq \cdots \)
\( stmt \, \text{bubble}_\text{dn} \triangleq \cdots \)

scope
\( \{ P_f \land 0 \leq n \} \) \( (a / \text{HEAP}[M](f, n) \parallel \text{true} \parallel S) \)
\( ~ (c / \text{ARRAY}[M](n) \parallel n = \#c \parallel (k / \text{INT} \parallel k = 0 \parallel S')) \)

exp
\( \{ \text{true} \} \) \( (f/a) : M ~ c_0 \)
\( \{ \text{true} \} \) \( (\#a) : \text{INT} ~ \#c \)

stmt
\( \{ \text{true} \} \) \( a := \{ \} \) ~ \( k := 0 \)
\( \{ \#a < n \land e \in a \} \) ~ \( c := (c; k : e') \); \( k := k + 1 \);
\( a := a \cup \{ \langle \text{exp} \, e \rangle : M \} \) ~ \( \text{bubble}_\text{up} \)
\( \{ \text{true} \} \) \( a := a - \{ f/a \} \) ~ \( c := (c; 0 : c_k) \); \( k := k - 1 \);
\( a := a - \{ v \} \)
\( \{ \text{true} \} \) \( \text{choose} \, v \, \text{so} \, v \in a \); ~ \( v' := c_k \); \( k := k - 1 \)
\( a := a - \{ v \} \)
\( \{ \text{true} \} \) \( \text{for}(v \parallel v \in a \parallel S) \) ~ \( \text{for}(i / \text{INT} \parallel 0 \leq i < k \parallel (v' / \text{true} \parallel v' := c_i ; S')) \)

Fig 10. Generic HEAP implementation module.
Proof Rules for Implementations

5.1. Overview

This chapter describes the role of implementation modules and directives in formal proofs of program correctness. We define a correct module as one whose translations respect couplings. For statement translations this requirement is defined in terms of a predicate transformer. The correct implementation of a nondeterministic statement can be characterized by a predicate transformer that is a generalization of the one for the implementation of deterministic statements. The proof that couplings are preserved can be conducted entirely independent of clients of the module and therefore need only be undertaken once.

Implementation directives in a program $S$ do not change the meaning of $S$. A process parallel to the determination of syntactic suitability of the implementation directives generates a set of verification conditions stated in terms of variables of $S$. If the verification conditions are met, the translation $S \rightarrow S'$ preserves the coupling of the free variables of $S$ to the free variables of $S'$. If the couplings of the free variables to their implementations are adequate relative to the specification of $S$, then $S'$ may be used in place of $S$.

Section 5.2 discusses notions of implementation, summarizing various implementation criteria, and presents the basis of our approach with the generalization of the original [Hoare72] criterion. Section 5.3 outlines proof rules for correct implementation modules. Section 5.4 gives the verification conditions incurred by an implementation directive and discusses the adequacy of couplings for variables whose occurrences are free in a program.

5.2. The Concept of Implementation

The simplest and most apparent requirement we can make of an "implementation" is that it behave identically to the abstraction for which it stands. The satisfaction of this requirement depends directly on what behavior we are able to observe and on what constitutes identical behavior. An implementation criterion is a guarantee that an implementation will exhibit the behavior of the abstraction on a certain class of observations.

As an example of the importance of the class of observations, consider an "abstract" program $S_a$ that is to be implemented by another program $S_c$. If we are satisfied to observe only the initial and final values of variables of $S_a$, an implementation $S_c$ could use a completely different algorithm but could still, as far as we could see, behave just like $S_a$. On the other hand, if we are allowed to observe the sequence of values assumed by variables of $S_a$ in the course of computation, then the possible implementations $S_c$ of $S_a$ are limited considerably.

An implementation criterion for this problem in a setting where intermediate values can be observed, but only initial and final values are required to be identical, was given in [Milner71] and was called a simulation. Roughly, Milner required that variables of $S_c$ had to be placed in correspondence with the variables of $S_a$ by means of a relation $R$. Corresponding steps in $S_a$ and $S_c$ had to preserve $R$. For the behavior of $S_c$ to be identical to $S_a$ on input and
output variables, \( R \) had to define a total function from input variables of \( S_a \) to input variables of \( S_c \) and from output variables of \( S_c \) to output variables of \( S_a \). By limiting observability to the input-output variables, Milner could settle for less stringent requirements on \( R \) between those variables that were not input or output variables (simply that \( R \) relate their values).

### 5.2.1. Implementation of data types

If we take our abstraction to be a data type, rather than a program, we have different definitions for what can be observed. A type definition introduces a new type \( s \), a set of operations \( O_s \) with result values of type \( s \), and a set of operations \( O_{\bar{s}} \) with result values of some type other than \( s \). The operations in the latter collection must have at least one parameter of type \( s \), the type of interest. The carrier of \( s \) defines the (abstract) values of type \( s \).

From the full collection of type definitions we can build up the set \( T \) of terms that involve a value of type \( s \) somewhere in the term. We partition \( T \) into \( T_s \), the terms that have type \( s \) and \( T_{\bar{s}} \), the terms that involve some operation of type \( s \) but have a result type that is not \( s \). The possible observations that can be made of an implementation of \( s \) are certainly limited to the terms in \( T \). It is possible to restrict \( T \) to terms involving only certain operations —this is the purpose of the collection of templates in a module. A further restriction of \( T \) limits our observation to instances of terms in \( T \) involving free occurrences of some identifiers that satisfy some predicate or pre-condition. If we satisfy an implementation criterion for a restricted set of terms \( T \), then we must insure that only terms in \( T \) can be observed —in our approach this is insured by the implementation suitability checks and by the verification conditions presented in section 5.4.

An implementation chooses representations for the carrier (set of abstract values) of each type in \( T \), and chooses manipulations of the representations for each of the operations in type \( s \). The implementation defines the relation between the representation and the abstract values of \( s \) with the coupling \( R_s \), and relies on the existence of couplings \( \bar{R}_{\bar{s}} \) between representations and abstract values of each type in \( \bar{s} \). If each operation of \( s \) establishes the coupling of its result value to the corresponding abstract value, assuming the representations of the operation arguments are coupled to their abstract values, then by induction, assuming all couplings \( \bar{R}_{\bar{s}} \) are preserved by operations not in \( s \), all terms in \( T \) are coupled to their abstract value.

In effect, what may be observed about the values of the terms in \( T \) is limited by the choice of couplings. For example, if the couplings \( \bar{R}_{\bar{s}} \) all specify the identity relation, then operations \( O_{\bar{s}} \) yield abstract values for terms in \( T_{\bar{s}} \), so that those terms behave identically under implementation. If \( R_s \) specifies a total function onto the abstract values, this defines an interpretation of observation of the \( T_s \) terms: the abstract value can be recovered from the observation. In analogy to the internal program variables of Milner's program simulation, if \( T \) is restricted to \( T_s \), observation of values of type \( s \) is precluded, and \( R_s \) may define any relation that can be maintained (or established) by the operations in \( O_s \).

An implementation criterion for implementations of data types will be relative to which sequences of operations of the type can be observed, and what can be observed about them. Historically, this "point of view" has been fixed in different ways in the major approaches to the definition of implementation criteria.
5.2.2. A comparison of implementation criteria

Simulation [Hoare72, Gries85, He86, Jones86].

Milner's simulation criterion between programs was a precursor to Hoare's criterion for
the implementation of an (abstract) data type with a SIMULA class. In terms of the discussion
of the previous section, Hoare's simulation criterion requires that operations in $O_S$ yield identi-
cal results under implementation, i.e. that the $\tilde{R}_S$ are identity relations, while $R_S$ relation must
specify a total function from the representation domain onto the abstract values; $R_S$ is known as
the abstraction function. This requirement suits the SIMULA setting well when programs res-
strict their use of classes to variables introduced in a scope; the input and output variables of
such a program are of a built-in type, and, under the simulation requirements, will have the
same value regardless of the choice of class for internal program variables. Directly related to
this approach are the APLHARD [Wulf76] and VDM [Jones80] criteria.

Early ideas in [Reynolds81], our own work [Gries85], the recent work by the PRG at
Oxford [He86] and [Jones86] all advocate the relaxation of the requirements on $R_S$. The goal in
[He86] and [Jones86] is to make the implementation criterion complete for nondeterministic
operations [Nipkow85]; our motivation is that such a generalization is required to develop par-
tial implementations, and we make no restrictions on either $R_S$ or the couplings $\tilde{R}_S$.

Algebraic Similarity [ADJ78, Wand79, Kamin83].

Another class of implementation criteria may be called algebraic, since they arise directly
from the algebraic definition of data types ([Guttag75, Liskov75]). Here, one algebra imple-
ments another algebra when there is a homomorphism between them. The direction of the
homomorphism depends on how an algebra is determined from a data type definition. An initial
algebra [ADJ78] lumps provably equal terms together, and the homomorphism to an imple-
menting algebra must keep these terms provably equal. A final algebra [Wand79, Kamin83]
distinguishes only between provably different terms; in this case the homomorphism is from the
implementing algebra and must not make equal terms distinct. In these approaches all terms
are considered observable and each must satisfy the defining equations of the corresponding
type. This is guaranteed to hold whenever there is a homomorphism in the proper direction
between the algebras.

The algebraic criteria are more restrictive than the simulation criteria since they make
stronger requirements of operations in $O_S$.

Behavioral Equivalence [Giarratana76, Sanella84, Schoett85, Nipkow85]

Some more recent work in the implementation of algebraic data types has moved towards
implementation criteria that partition the set of types into "observable" types $V$ and "hidden"
types $\tilde{V}$. Only terms of an observable type need have values equivalent under implementation,
which means that for $t \in V$, $R_t$ is the identity coupling; the remaining couplings are uncon-
strained.

The original simulation criterion of Hoare is generalized by the behavioral equivalence
criterion (choose the only hidden sort to be the type $s$ of interest), since $R_s$ need not be an
abstraction function. The algebraic criterion is also generalized by behavioral equivalence since
it is possible to define algebras that are behaviorally equivalent but not isomorphic [San-
ella84].
Partial Implementation [Hoare72, Kamin84, Gries85, Schoett85]

Hoare [Hoare72] gives an example implementation of an integer set \( a \) with the elements \( a'[1..m] \) where \( a' \) is an array of length 100. The abstraction function can not be preserved over the operation \( a := a \cup \{v\} \) when \( #a = 100 \), so that the implementation appears not to be adequate. Since all (machine) implementations of mathematical abstractions will encounter this problem eventually, it is necessary to include a provision in the implementation criterion to address this problem.

Hoare indicates that the abstract and concrete values can be restricted by two data type invariants that define the range and domain, respectively, on which the coupling defines a total and onto function. In his example, the abstraction function is preserved by the implementation for set insertion only when \( #a < 100 \), which becomes a verification condition for a client using the implementation. In an algebraic setting, [Kamin84] generalizes the implementation criterion to be a partial homomorphisms. The concept is treated in a different fashion in [Schoett85] where an algebra \( A \) is behaviorally weaker than \( B \) if \( A \) can either fail or be equivalent to \( B \). In these cases no preconditions are explicitly identified, so that the verification that a program can use a behaviorally weaker implementation would be difficult.

Our only requirement is that couplings be preserved by implemented operations. If, for example, it is not possible to test (observe) equality between two sets of integers, then there is little point in requiring that an implementation represent distinct sets distinctly. Or, as an academic example, if it is not possible to observe anything about a data type (e.g. some 'write-only' type), then why relate a representation to any abstract value at all? The existence of abstract values that have no representation in an acceptable implementation is described as an error in the data-type specification in [Jones80]. Such abstract values can not be observed with the implementation, hence the specification is 'biased' and should be fixed to eliminate the offending states. The argument is that there is a better specification for which the partial implementation represents all states.

But our point of view is that data type specifications should be fixed and mathematical at the outset of program development. We should not have to go back and redefine a data type with a specification that admits a more efficient implementation, because our derivation will have to be redone and the specification is difficult. Instead, by making no requirements on the form of the couplings, we can give a definition of implementation that is insensitive to bias introduced by a particular usage pattern of high-level data types, allowing us to develop partial implementations geared to a particular use of the notation. We can develop such a criterion by generalization of the simulation criterion of [Hoare72], which is what we do next.

5.2.3. Generalization of the simulation criterion

For the time being we postpone the question of adequacy of an implementation module for a client program and concern ourselves only with the correctness of the module. We start with the [Hoare72] correctness criterion for a form of implementation based on the SIMULA class. A class \( T \) corresponds to a type with operations \( p_0, \ldots, p_{m-1} \). A program variable \( a \) declared to have type \( T \) may be operated on only with these operations. The class \( T \) consists of concrete variables \( \tilde{c} \) that represent the value of the abstract variable, an initialization of the \( \tilde{c} \), and, for each operation \( p_i \), the signature of \( p_i \) and its implementation. All operations in \( T \) implicitly have an abstract variable of type \( T \) as an argument that is instantiated in a client program.
as a prefix to the operation name. The operations may be procedures or functions. For example (with some notational adjustments), the class IntSet would contain the operations

\[
\text{proc insert}(i : \text{Int}); (\text{Stmt } S) \\
\text{func has}(i : \text{Int}); (\text{exp } e)
\]

The corresponding invocations in a client on a variable \(a\) of type \(T\) are the \textit{statement} \(a.\text{insert}(\langle \text{exp} \rangle)\) and the \textit{expression} \(a.\text{has}(\langle \text{exp} \rangle)\). The implementations \(S\) and \(e\) may use \(i\) and \(\overline{c}\) and may assign to \(\overline{c}\) (when this occurs in \(e\), this is a “benevolent” side-effect).

The correctness of a class is proved in a number of steps.

(1) **Determine a model for the operations.**

For each operation find a function that describes the change to \(a\) made by a procedure operation or describes the value of the operation if it is a function. The model for class IntSet is, of course, sets of integers, and these will be the values over which \(a\) ranges. The intent of the model is to provide an axiomatic definition of the operations of the class that can be used in the development of an abstract program. An axiomatic definition can be given via the assignment rule if the operations can all be expressed as functions and updates to abstract variables. For the two operations above, the definitions are

\[
a.\text{insert}(i) \triangleq a := a \cup \{i\} \\
a.\text{has}(i) \triangleq i \in a
\]

For example, the first definition gives a predicate transformer for the \textit{insert} operation:

\[
\begin{align*}
\text{wp.}(\langle a.\text{insert}(i) \rangle', R) \\
\quad = \text{wp.}(\langle a := a \cup \{i\} \rangle', R) \\
\quad = R^a_{\Delta \{i\}}
\end{align*}
\]

Non-deterministic operations can not be modeled by simple assignments (because non-deterministic expressions are not referentially transparent, substitution is not well defined), so that Hoare’s rule does not extend to nondeterministic operations. The extension to procedure operations that alter arguments other than the implicit argument \(a\) presents technical complications in this approach and was not further considered in Hoare’s paper.

(2) **Choose correct data type invariants.**

Choose predicates \(I_c\) and \(I_a\) to restrict the possible values of \(c\) and \(a\), respectively. Show that they are invariant across all operations on \(a\) and \(c\) respectively. That is, for each operation body \(S_c\) in the class IntSet show

\[
\{ I_c \} S_c \{ I_c \}
\]

and for each model \(f\) of an abstract operation show

\[
I_a(a) \Rightarrow I_a(f(a))
\]

Since the data type invariants are maintained by all possible operations on \(a\) and \(c\) including initialization operations, they limit the values that \(a\) and \(c\) can assume.

(3) **Choose a correct abstraction function.**

The goal here is to find a function \(h\) from concrete values (of \(c\)) to abstract values (of \(a\)). The function has to be total on the domain \((c \parallel I_c \parallel c)\) and onto the range \((a \parallel I_a \parallel a)\); i.e. the abstract value can be \textit{retrieved} from the concrete value. We have to prove that each concrete
operation \( S_c \) respects the corresponding function \( f \) in the model as viewed through the abstraction function:

\[
\{ t = h(c) \} \quad \{ f(t) = h(c) \} \tag{5.1}
\]

where \( t \) is a logical variable that denotes the abstract value on which \( f \) operates.

These three steps complete Hoare's requirements. We now continue with a reformulation and generalization. First, instantiate \( t \) in (5.1) by the actual abstract variable \( a \) and record the operation on \( a \) as \( a := f(a) \) to obtain

\[
\{ a = h(c) \} \quad \{ f(a) = h(c) \} \quad a := f(a) \quad \{ a = h(c) \} \tag{5.2}
\]

This establishes the invariance of the abstraction function under juxtaposition of the abstract operation and its implementation, which is meaningful because our programming notation should include both operations.

In addition to the maintenance of the abstraction function, the implementation preserves \( I_c \) and the abstract operation preserves \( I_a \). If we take the set of variables operated upon by the model and the implementation to be disjoint (which is not the case in Hoare's treatment, but always holds for our translations), we can include both of the representation invariants in (5.2) to arrive at

\[
\{ I_a \land I_c \land a = h(c) \} \quad \{ f(a) \land I_a \land I_c \land a = h(c) \}
\]

Now we make some generalizations. We relax the requirement that \( h \) be an abstraction function and use a more general relation \( R(a, c) \). If we let \( R_{impl} = R(a, c) \land I_a \land I_c \) then we have a single coupling (of \( a \) to \( c \)) that is preserved by the abstract and concrete operation taken together. In general we can implement any statement \( S_a \), not just statements of the form \( a := f(a) \) (although for non-deterministic statements the juxtaposition is too restrictive), and we can choose a precondition \( P_M \) to limit the states in which the implementation criterion must hold. We know \textit{a priori} that the implementation criterion can be restricted to the states \( wp.(S_a, true) \), since a correct program can not exercise an implementation outside of that domain. Therefore, \( S_c \) is a correct implementation of \( S_a \) under coupling \( R_{impl} \) and restriction \( P \) when

\[
R_{impl} \land P \land wp.(S_a, true) \Rightarrow wp.(S_c, S_a, R_{impl})
\]

The coupling is a single predicate which is easy to work with and the implementation criterion is a single simple rule that extends easily to include other variables in the operations, nondeterminism in \( S_a \) and \( S_c \), and cooperating implementations. It has a simple analog for the implementation of expressions that does not involve a predicate transformer. Since we demonstrated a coupling that satisfies our criterion for any SIMULA class shown correct according to the rules in [Hoare72], our rule generalizes that criterion.

5.3. Module Correctness Rules

Assume module \( M \) represents abstract variable \( a \) with concrete variable \( c \) under coupling \( I_M \). The module is correct if each of its implementation units preserves couplings. The meaning of "preserves" varies with the class of the implementation unit.

In any implementation unit, the couplings that may be assumed can be systematically determined. An implementation unit may contain a number of nonterminals in the template.
For each \((\text{exp } e) : E\) or \((\text{id } e) : E\) in the template, the value \(e\) is coupled to the value \(e'\) by \(I_E\). If \(E\) is omitted, we shall take the coupling of \(e\) to \(e'\) to be \(e = e'\). For each parameter \(f : F\) of \(M\) used in the implementation unit, \(f\) is coupled to its representation \(f'\) by \(I_F\). If \(F\) is omitted, we take the coupling to be \(f = f'\). By assumption, a nonterminal \((\text{stmt } S)\) maintains the couplings of all variables that occur free within it when it is juxtaposed with its implementation \(S'\). We do not know which variables occur free in \(S\) since it may stand for any statement, but because the translation is syntax directed, the coupling of any variable that occurs free in the template will be preserved by \(S'; S\). For example, unless the class of the implementation unit is a scope, the variable \(a\) can occur free in a template with a nonterminal \((\text{stmt } S)\), so that we may use \(I_M \Rightarrow \text{wp.}(S'; S, I_M)\). Let \(I^*\) stand for the conjunction of the couplings of the nonterminals, the parameters, and the predicates generated from \((\text{stmt})\) nonterminals. The predicate \(I^*\) contains our assumptions that are independent of implementation unit class. The correctness requirements of an implementation unit are now presented by class.

Expressions.

Let the implementation unit be \(\{P\} \hat{S}_a : N \sim e_c\), and let \(e_a\) be the sentential form of \(\hat{S}_a\). The abstract and concrete expressions must yield values coupled by \(I_N\), so the rule is

\[
I^* \land I_M \land P \land \text{dom. } e_a \Rightarrow I_N(e_a, e_c) \tag{5.3}
\]

Statements.

Let the implementation unit be \(\{P\} \hat{S}_a : S_c\) and let \(S_a\) be the sentential form of \(\hat{S}_a\). The juxtaposition \(S_c; S_a\) must preserve the couplings of all variables that occur free in \(S_a\). Since \(f_{\hat{S}_a}(S_a) \subseteq f_{\hat{S}_a}(I^* \cup I_M)\) (we assume each abstract variable is represented with precisely one coupling), it suffices to require that

\[
I^* \land I_M \land P \land \text{wp.}(S_a, \text{true}) \Rightarrow \text{wp.}(S_a; S_c, I^* \cup I_M) \tag{5.4}
\]

In analogy to the rule for the implementation of expressions, we shall refer to the consequent of (5.4) as the predicate \(I_{\text{stmt}}\). The syntactic properties of the programming notation described in section 2.3.2 allow us to conclude most of the couplings in \(I^* \cup I_M\) are preserved simply because their variables are not in \(\text{def.}(S_a)\). Hence (5.4) may easily be specialized to establish only the couplings for variables defined in \(S_a\). Often this will reduce \(I_{\text{stmt}}\) to \(\text{wp.}(S_a; S_c, I_M)\).

Scopes.

The requirement for the implementation of scopes is really a particular case of the requirement for statements, since scopes are statements. The difference is that \(a\) does not occur free in the scope that introduces \(a\), hence we do not assume or preserve \(I_M\), but only require that it be established. Let \(S_a\) be the scope template \((a \parallel A \parallel (\text{stmt } S))\), and let \(S_c\) be its implementation \((c \parallel C \parallel S')\). A sufficient condition for correctness of the implementation under precondition \(P\) is:

\[
I^* \land P \land \text{wp.}(S_a, \text{true}) \Rightarrow \forall(a, c \parallel A \land C \parallel I_M) \tag{5.5}
\]

5.3.1. Simulation of nondeterministic statements

We have assumed that our abstract operations \(S_a\) are deterministic. If they are nondeterministic, then (5.5) is too restrictive. We illustrate the problem with the following example. Consider the implementation of \(a : \text{Int} \sim c : \text{Int}\) with coupling \(a = c\). If we investigate an
implementation $S'$ for the statement

$S$:  choose $a$ so $a > 0$

we find that we must have

$$\{a = c\} S';  \text{choose } a \text{ so } a > 0 \{a = c\}$$

$$\equiv \{a = c\} S' \{ \forall (a \parallel a > 0 \parallel a = c) \}$$

$$\equiv \{a = c\} S' \{ \text{false} \}$$

any solution for $S'$ would violate the excluded miracle unless we choose the implementation precondition to be false, which would not be helpful. Hence we are led to the conclusion that no implementation $S'$ exists!

Our view of nondeterminism in program development is "demonic": correct nondeterministic programs satisfy their specification regardless of the nondeterministic choices made. Therefore an implementation $S'$ of $S$ should be free to choose a value for $c$ that may be coupled to some choice for $a$ satisfying $a > 0$, rather than be required to choose a value for $c$ that is coupled to every possible choice for $a$. For example, acceptable implementations of $S$ (under the coupling above) should include $c := 3$ and even choose $c$ so $c > 0$.

To reflect this less restrictive interpretation of implementation of nondeterministic statements, we require that execution of $S'$, when started in a state satisfying $a = c$ must terminate in a state in which execution of choose $a$ so $a > 0$ has the possibility to re-establish $a = c$.

We define the set of states in which a statement $S$ has the possibility to terminate in a state satisfying predicate $R$ to be given by an "angelic" predicate transformer $\text{wp.}(S,R)$. Any state in which $S$ is guaranteed to terminate and in which it is not the case that execution of $S$ is guaranteed to establish $\neg R$, must be a state in which $S$ has the possibility to establish $R$. Consequently we have that

$$\text{wp.}(S,\text{true}) \land \text{wp.}(S,R) \equiv \text{wp.}(S,\text{true}) \land \neg \text{wp.}(S,\neg R)$$

This particular characterization of $\overline{\text{wp}}$ in terms of $\text{wp}$ is perfectly suited to our needs because the states in which the abstract statement $S$ is coupled to $S'$ are limited to states in which $S$ terminates. Therefore we can rephrase proof obligation (5.4) for the implementation of statements as

$$I_{\text{stmt}}(S,S') \equiv \text{wp.}(S',\neg \text{wp.}(S,\neg I^*))$$

(5.6)

where $I^*$ represents the conjunction of all couplings for variables that occur free in $S$. The statement choose $c$ so $c > 0$ is a valid implementation of choose $a$ so $a > 0$ using definition 5.6 with coupling $a = c$ since

$$\text{wp.}(\text{choose } c \text{ so } c > 0, \neg \text{wp.}(\text{choose } a \text{ so } a > 0, \neg (a = c)))$$

$$\equiv \forall (c \parallel c > 0 \parallel \neg (a \parallel a > 0 \parallel \neg (a = c)))$$

$$\equiv \forall (c \parallel c > 0 \parallel \exists (a \parallel a > 0 \parallel a = c))$$

$$\equiv \text{true}$$

which is certainly implied by $a = c$. When $S$ is deterministic, (5.6) is equivalent to (5.4).

5.3.2. Multiple implementation problem

We restricted each variable to be implemented by exactly one module. This is necessary because multiple implementations may make different (incompatible) observations about the
resolution of non-determinism in operations. We illustrate this with an example.

Suppose modules M1 and M2 both implement the nondeterministic statement S: choose a so a > 0 of the previous section with couplings

\[ I_{M1}(a, c_1) \models a = c_1 \text{ and} \]
\[ I_{M2}(a, c_2) \models a = c_2 \]

but that M1 implements S as \( c_1 := 1 \), while M2 implements the same statement as \( c_2 := 2 \). Now consider the "multiple" implementation of a variable a by both modules, i.e.

\[ a / M1, a / M2 \vdash S \sim S' \]

This means that a is coupled to two representations, \( c_1 \) and \( c_2 \), and both couplings must be preserved across the implementation \( S' \) of S. Coupling (5.6) must be established taking \( S' \) to be \( c_1 := 1; c_2 := 2 \), and \( I^* \) to be \( I_{M1} \land I_{M2} \). Simple substitution shows that we must obtain

\[ \exists(a \parallel a > 0 \parallel a = 1 \land a = 2) \]

which is false and hence cannot be established. Clearly S can choose to preserve \( I_{M1} \) or \( I_{M2} \), but if the two concrete operations yield representations of two distinct values for the abstract variable a, S cannot choose to act like both. This leads us to conclude that, in the presence of non-deterministic operations, we cannot have a variable implemented more than once.

5.4. Directive Correctness Rules

The use of a correct implementation module M in a correct program S is not sufficient to insure that the translated program \( S' \) will meet the specification of S. There are two reasons for this. First, an implementation directive for a variable that occurs free in S means that \( S' \) operates on a representation which may fail to represent some abstract values of interest. The values of interest are not determined by other observing operations, in this case, but by the context in which the program is placed. A rule to guarantee that the implementations of variables that occur free in S are adequate, independent of context, is the subject of the next section. In this section we concern ourselves with the adequacy of implementations within the context of the client program. An implementation directive that is syntactically suitable within a client does not preclude an implementation of an operation being used outside of its domain of applicability, given by the associated unit precondition. Therefore we divide the complete proof obligation into a module-correctness proof formulated on the module in isolation, and an integration proof that is specific to a client of the module.

The integration proof is phrased purely in terms of preconditions on abstract values, so it fits in directly with the notation of a given client. With each application of an inference rule in the formal system to translate programs, the implementation unit corresponding to the rule requires a precondition to hold in the client program. The precondition is adapted from its form in the implementation module to a predicate on client variables and expressions by the instantiation of the module with the directive. Each of these preconditions needs to be established from the client precondition preceding the first \( \langle stmt \rangle \) production at or above the matched program fragment.

The translation of a large program could entail many cumbersome verification conditions, particularly if the implementations use very limited representations. Unit preconditions in a module that are true are, of course, always met and do not lead to verification conditions. We
have mentioned two techniques (strengthening the module coupling to eliminate repeated verification of a recurring condition and implementation of nondeterministic statements) that can be used to simplify module preconditions or reduce them to true.

One other possibility is to try to include the preconditions into the client's proof outline, but this is quite difficult and could potentially alter the client so that it no longer satisfies its specification. At the current time we foresee that programmers, presented with verification conditions upon the inclusion of implementation directives in a program, investigate a variety of adjustments to the program proof in order to achieve the required conditions, or perhaps abandon one directive in favor of another.

The next section presents one more technique that can always be used to simplify the integration proof, or eliminate it altogether, if desired.

5.4.1. Weak implementation correctness

We can introduce a "miraculous" statement fail that may be used only on the implementation side of a module. This allows us to give two interpretations of the implementation criterion — total and weak correctness. For the strong criterion we define \( wp.(\text{fail}, R) = false \), so that fail can never be used to achieve a result. For the weak criterion we take \( wp.(\text{fail}, R) = true \), so that fail performs "miracles". The semantic effect of fail is to yield failure of the program. For example, in the implementation unit

\[
\{0 \leq v < n\} \quad a := a \cup \{v\} \quad c(v') := true
\]

in the BITV implementation is totally correct, but has a precondition that is not identically true, while the implementation unit

\[
\{true\} \quad a := a \cup \{v\} \quad \text{if } v' < n' \rightarrow c(v') := true \quad \text{if } v' \geq n' \rightarrow \text{fail}
\]

is weakly correct and will lead to an error if an insertion outside of the representation bounds were to be attempted, but has precondition true.

Modules developed with fail statements are "safer" if no integration proofs are performed, but such modules will likely lead to less efficient operation. Of course the programmer must be prepared to receive "failure" as a (possible) result when using weakly correct modules.

5.4.2. Adequacy of representations for free variables

The module correctness rules and the integration proofs assure us that for each translated fragment, couplings are respected. Suppose we have a program \( S \) fully annotated with syntactically suitable implementation directives, including one for the free variable \( a \), satisfying

\[
a / \hat{M} \vdash S \sim S'
\]

(5.7)

If (1) \( S \) is correct relative to a specification \( (P(a), Q(a)) \), i.e. \( P(a) \Rightarrow wp.(S, Q(a)) \), and (2) the implementation modules used in \( S \) are correct, and (3) the implementations are syntactically suitable for \( S \), and (4) \( S \) satisfies the verification conditions generated in the translation, then

\[
[I(a, a')] S; S' \{I(a, a')\}
\]

(5.8)
where \( I \) is the coupling of module \( M \). Since \( S' \) involves no abstract variables we can strengthen (5.8) to

\[
\{ P(a) \land I(a, a') \} \implies S'; \quad \{ S' \land Q(a) \land I(a, a') \} \quad (5.9)
\]

The implementation \( S' \) is adequate relative to the coupling of the free variable \( a \) to the implementation \( a' \). However, the coupling for \( a \) is not determined solely by operations in \( S \); which values of \( a \) must be represented depends on the context in which \( S \) is employed, which is not known to us. For example, if \( S \) built a set \( a \) to satisfy the postcondition \( a = \{ 1..10 \} \) using only insert operations and without further reference to \( a \), then \( M \) might specify a very poor coupling \( I \) such as \textit{true} or \textit{false}, yet provide correct translations since nothing about \( a \) is observed in \( S \). Clearly the notion of adequacy for free variables needs to be extended to consider the program context.

Ultimately, our objective is to use the translation \( S' \) in place of \( S \). That is, we want to make sufficient requirements of \( I \) such that for any \( a \) satisfying \( P(a) \) we can use \( I \) and \( S' \) to find an \( \hat{a} \) satisfying \( Q(\hat{a}) \). Two requirements on \( I \) relative to the specification \( (P(a), Q(a)) \) are sufficient to insure this.

First we require that every abstract value \( a \) satisfying the precondition can be coupled to some representation \( a' \) by \( I \):

\[
\forall (a \parallel P(a) \parallel \exists (a' \parallel \textit{true} \parallel I(a, a'))) \quad (5.10)
\]

The second requirement is that all abstract values coupled to the same representation act equivalently in the postcondition. That is, a concrete value \( a' \) may be coupled to several abstract values \( a \), provided that \( Q(a) \) is either true for all \( i \) or false for all \( i \). To be precise,

\[
\forall (a_1, a_2, a' \parallel I(a_1, a') \land I(a_2, a') \parallel Q(a_1) \iff Q(a_2)) \quad (5.11)
\]

To see that these requirements are sufficient, suppose we have some \( a \) satisfying \( P(a) \). Now (5.10) guarantees that there is some \( a' \) for which \( I(a, a') \) holds; any such \( a' \) will suffice. With \( a \) and any \( a' \) we satisfy \( P(a) \land I(a, a') \), so that we conclude that execution of (5.9) will terminate in a state where the abstract value \( b \) and the concrete value \( b' \) satisfy \( Q(b) \land I(b, b') \). Since \( b \) satisfies \( Q(b) \) and \( b \) is coupled to \( b' \) we have by (5.11) that any \( \hat{a} \) such that \( I(\hat{a}, b') \) will satisfy \( Q(\hat{a}) \). Consequently it is not necessary to know the value \( b \) of the abstract variable; any abstract value coupled to \( b' \) is sufficient. This makes \( b \) a ghost variable in execution of \( S' \), \( S \) so that \( S \), the abstract program, need not be executed.

A quick inspection of the requirements yields some insight into the adequacy of the extremal couplings \textit{true} and \textit{false}. The coupling \textit{true} is adequate only for specifications \( (\textit{true},\textit{true}) \) and \( (\textit{true},\textit{false}) \), with the former satisfied by any terminating program whatsoever and the latter unsatisfiable. The coupling \textit{false} is only adequate with a specification of the form \( (\textit{false},Q) \) which is satisfied by a program which can not be started in any state. This is the proper state of affairs because with such peculiar couplings almost any implementation can be shown correct.

This concludes the section on adequacy and ends this chapter on the formal rules governing the use and meaning of implementation modules and directives.
6 Concluding Remarks

We have described an approach to programming that can be conducted entirely within a high-level specification language. Like many others, we favor mathematical notations for the specification and development of programs. Partial implementations of the notations are both necessary and desirable. Necessary because they form a realistic account of implementation; one that can be used to develop correct programs for actual computing machinery. Desirable because notations, liberated from concerns of overall efficiency, may be chosen primarily for their expressiveness and simple theory and programs may be developed with such notations, without a priori commitments to a class of implementations.

Our view of programming (but not exactly our approach to it) is inspired by Hoare’s "relational programming" paradigm [Hoare85] in which programs are identified as a particular subset of specifications having a certain structure (they are composed of simple relations that have an executable interpretation). The selective rewriting of specifications to place them in the subset of specifications that are programs is accomplished by application of the theory of the notation. An analogy that is not too far-fetched is that of algebraic rearrangement of an arithmetic proposition into a form where one variable is expressed in terms of all the others.

What we feel is lacking in this view is a sense of continuity in program development: Hoare fixes the space of programs so that until the specification is rewritten completely into that space, there is no program at all. Moreover, Hoare’s space of programs (which is the only subset he recommends) is extremely low-level and requires a great deal of refinement, even for the simplest of specifications (it is, of course, a marvelously efficient space of programs on current machinery). But this approach excludes the rapid prototyping paradigm.

A collection of partial implementations of a notation provides us with many overlapping subsets of specifications that correspond to programs. Since we expect that all type specifications will have some very general (but perhaps weak) implementations, in principle many specifications may be directly executed. We might use some search-strategy (or more likely a theorem prover) to find values for variables satisfying certain kinds of specifications, assuming enough patience while waiting for results. More realistically, we have to perform some refinement, but ultimately we hope that the effort required might be proportional to final efficiency rather than to the complexity of the specification.

By fixing on a notation for a problem area, rather than defining abstract data types de novo with the specification of a program, we hope, first of all, to be able to "get good at" the refinement and rewriting process, and second of all to take advantage of the essential stability of the notation to build up and use libraries that encapsulate, in the form of partial implementations, the more cumbersome refinement of one data type into another.

It might be argued that our fixed notations force abstractions on people. This is not strictly true; it is always possible to define a new notation and give a variety of partial implementations for it. But it is a lot of work. Thus it is possible that this approach limits how new concepts can be expressed and manipulated. If the experiences with high-level programming languages like
APL and SETL are any indication, an expressive and familiar notation greatly improves the reliability and (dare we say it) productivity of the programmer, even when the problem is not particularly suited to the abstractions available. The principal impediment to the success of these languages is the lack of a path to improved efficiency of the final product, should such a requirement develop. This is precisely what we have tried to address with our notions of encapsulation and implementation.

The next step is the application of these techniques to see whether they confer the claimed advantages in the development of programs. This will involve both the definition of very general notations and the development of a substantial library of partial implementations. We are looking forwards to it.
Appendix  Type Definitions

A.1. Overview

This appendix gives three type definitions that introduce the notation used in examples throughout the text. The notation here is a part of a more general notation under development for a handbook of algorithms in preparation by Gries. Most functions introduced should need little description, since they are standard functions in the mathematical literature. More unusual functions are briefly described with each definition. No effort has been made to present the formal properties of the notation other than the inclusion of some references. In the case of statements, the predicate transformers are presented since they are unique to these particular definitions.

The type definitions for $\text{Bool}$, $\text{Set}(\sigma)$ and $\text{Seq}(\sigma)$ are given below; the definition for $\text{Int}$ has been omitted.

A.2. Type $\text{Bool}$

Type $\text{Bool}$ is defined in Fig. 12. The statements introduced with type $\text{Bool}$ are a generalization of the alternative and repetitive construct in the guarded command language of [Dijkstra75]. They are generalizations because each guarded command is a scope that may introduce variables, with values chosen to satisfy the guard, that extend over the statement. The predicate transformers for the two statements can be defined as follows.

Let $\text{IF}$ stand for the alternative construct and $\text{DO}$ for the repetitive construct in the definitions above, and let $n$ be the number of guarded commands in the construct. For any $R$:

\[
wp(\text{IF}, R) = \exists (i \parallel 0 \leq i < n \parallel \exists (\nu_i \parallel B_i)) \land \forall (i \parallel 0 \leq i < n \parallel \forall (\nu_i \parallel B_i \parallel wp. (S_i, R)))
\]

In words, there exists some guard $B_i$ that can be satisfied by some choice of values for the associated $\nu_i$, and for any guard, under any choice of values that satisfies the guard, the execution of the associated command is guaranteed to terminate in a state satisfying $R$.

The predicate transformer $wp(\text{DO}, R)$ can not be so succinctly characterized; the usual approach is to give sufficient conditions to conclude that a loop invariant $I$ is a precondition of $\text{DO}$ for a given $R$. If a predicate $I$ is invariant across each guarded command,

\[
\forall (i \parallel 0 \leq i < n \parallel I \Rightarrow \forall (\nu_i \parallel B_i \parallel wp. (S_i, I)))
\]

and with all guards false $R$ is established from $I$,

\[
I \land \forall (i \parallel 0 \leq i < n \parallel \neg \exists (\nu_i \parallel B_i)) \Rightarrow R
\]

and $\text{DO}$ can be shown to terminate (see, e.g. [Gries81]), then $I$ is a loop invariant, and a precondition of the loop with postcondition $R$:

\[
I \Rightarrow wp. (\text{DO}, R).
\]
### Type `Bool`

<table>
<thead>
<tr>
<th>exp</th>
<th>Bool × Bool → Bool</th>
<th>~ p ∧ q</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>Bool × Bool → Bool</td>
<td>~ p ∨ q</td>
</tr>
<tr>
<td>or</td>
<td>Bool × Bool → Bool</td>
<td>~ p → q</td>
</tr>
<tr>
<td>implies</td>
<td>Bool × Bool → Bool</td>
<td>~ p ⇔ q</td>
</tr>
<tr>
<td>equiv</td>
<td>Bool × Bool → Bool</td>
<td>~ p = q</td>
</tr>
<tr>
<td>not</td>
<td>Bool → Bool</td>
<td>~ ¬ p</td>
</tr>
<tr>
<td>forall</td>
<td>Scope(Bool) → Bool</td>
<td>∀ x X</td>
</tr>
<tr>
<td>exists</td>
<td>Scope(Bool) → Bool</td>
<td>∃ x X</td>
</tr>
<tr>
<td>true</td>
<td>→ Bool</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>→ Bool</td>
<td>false</td>
</tr>
<tr>
<td>eq</td>
<td>α × α → Bool</td>
<td>~ x = y</td>
</tr>
<tr>
<td>noteq</td>
<td>α × α → Bool</td>
<td>~ x ≠ y</td>
</tr>
</tbody>
</table>

#### stmt

\[
\begin{align*}
& (\text{if } \{(\text{scope } \langle \text{id} \overline{v}_i \rangle \langle \text{exp } B_i \rangle \langle \text{stmt } S_i \rangle)\}^i : \text{Scope(Stmt)}) \\
& \quad \text{~ if } (\overline{v}_i \parallel B_i \rightarrow S_i)^i \text{ fi} \\
& (\text{do } \{(\text{scope } \langle \text{id} \overline{v}_i \rangle \langle \text{exp } B_i \rangle \langle \text{stmt } S_i \rangle)\}^i : \text{Scope(Stmt)}) \\
& \quad \text{~ do } (\overline{v}_i \parallel B_i \rightarrow S_i)^i \text{ od}
\end{align*}
\]

#### props

See [Manna85], [Gries82] and text.

---

### A.3. Type `Set(σ)`

The type definition for sets is given in Fig. 13. Most of the set operations are standard and we clarify only a few of the less familiar functions. The size of a set (#s) is the number of elements in it; the complement is relative to the universe of elements of the same type $s : \text{Set(σ)} \vdash \overline{s} = \{ v : σ \parallel v \in s \parallel v \}$. Functions construct and reduce are defined by example in section 1.8, and $f^*s = \{ v : v \in s \parallel f.v \}$. Function `setoerset` is defined by the identity `(t) = \{ i : 0 \leq i < \#t \parallel t_i \}`.

The `iterate` statement is a generalized iterator that repeats $S$ with successive values $\overline{v}$ that satisfy $P$. No order is implied. Let $T$ stand for the statement `foreach(\overline{v} \parallel P \parallel S).` Using a set $s$ we can give the meaning of $T$ as the refinement

\[
\begin{align*}
(s \parallel s = \{ \}) \parallel \\
\quad \text{do } (\overline{v} \parallel P \land (\overline{v}) \not\in s \rightarrow S; s := s \cup (\overline{v}) \}) \text{ od}
\end{align*}
\]

(for some $s \not\in (fv.P \cup fv.S)$). We constrain `def.S \cap \overline{v} = ∅` to ensure that $S$ can not alter the bound variables $\overline{v}$, and also constrain `fv.P \cap def.S = ∅` to ensure that the set of values $V = (\overline{v} \parallel P \parallel (\overline{v}))$ (the values characterized by $P$) is invariant under $S$. Under these restrictions
**type** Set(σ)

**exp**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>→ Set(σ)</td>
<td>~ { } or Ø</td>
</tr>
<tr>
<td>singleton</td>
<td>σ → Set(σ)</td>
<td>~ { a }</td>
</tr>
<tr>
<td>member</td>
<td>σ×Set(σ) → Bool</td>
<td>~ a ∈ s</td>
</tr>
<tr>
<td>union</td>
<td>Set(σ)×Set(σ) → Set(σ)</td>
<td>~ s ∪ t</td>
</tr>
<tr>
<td>intersect</td>
<td>Set(σ)×Set(σ) → Set(σ)</td>
<td>~ s ∩ t</td>
</tr>
<tr>
<td>difference</td>
<td>Set(σ)×Set(σ) → Set(σ)</td>
<td>~ s − t</td>
</tr>
<tr>
<td>complement</td>
<td>Set(σ) → Set(σ)</td>
<td>~ s̄</td>
</tr>
<tr>
<td>construct</td>
<td>Scope(σ) → Set(σ)</td>
<td>~ { X }</td>
</tr>
<tr>
<td>reduce</td>
<td>(σ×σ → σ)×Set(σ) → σ</td>
<td>~ f / s</td>
</tr>
<tr>
<td>each</td>
<td>(β → σ)×Set(β) → Set(σ)</td>
<td>~ f &quot;s</td>
</tr>
<tr>
<td>seqtoSet</td>
<td>Seq(σ) → Set(σ)</td>
<td>~ { t }</td>
</tr>
<tr>
<td>subset</td>
<td>Set(σ)×Set(σ) → Bool</td>
<td>~ s ⊆ t or t ⊇ s</td>
</tr>
<tr>
<td>size</td>
<td>Set(σ) → Int</td>
<td>~ #s</td>
</tr>
</tbody>
</table>

**stmt**

\[
(\text{iterate}\ (\text{scope}\ \langle\text{idl}\ \bar{v}\rangle\ \langle\text{exp}\ P\rangle\ \langle\text{stmt}\ S\rangle) : \text{Scope(Stmt)})
\]

~ foreach(\bar{v} \parallel P \parallel S)

**props**

See [Manna85] and text.

---

the termination of (A.1) depends directly on whether P is finitary and whether we can find a predicate I that satisfies

\[
\forall (s, \bar{v} \parallel s \subseteq V \wedge \bar{v} \in V \wedge \bar{v} \in s \parallel I \Rightarrow \text{wp}(S, l^p_{\bar{v}}(\bar{v})))
\]

If \( l^p_{\bar{v}} \Rightarrow R \), then \( l^p_{\bar{d}} \Rightarrow \text{wp}.(T, R) \) and \( l^p_{\bar{d}} \) is a precondition of T with postcondition R.

**A.4. Type Seq(σ)**

The type definition for sequences is given in Fig. . We view a sequence as a function from integers to values of an arbitrary type. The operations that may not be familiar are the formation of a sequence of length 1 from an element a as [a]; the pointwise update \((t; v : w)\) which is a sequence that agrees with t everywhere except at index v where its value is w; the selection \(s[t]\) of values from sequence s in the order of the indices given in the integer sequence t, satisfying \(s[t] = [i \parallel 0 \leq i < \#t \parallel v(t_i)]\).

The ordered iterator can be defined as

\[
\text{for}(v \parallel v \in t \parallel S) \triangleq (i \parallel i = 0 \parallel \text{do } i < \#t \rightarrow (v \parallel v = t_i \parallel S) \text{ od })
\]
type $\text{Seq}(\sigma)$

<table>
<thead>
<tr>
<th>exp</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{empty}$</td>
<td>$\to \text{Seq}(\sigma)$</td>
</tr>
<tr>
<td>$\text{single}$</td>
<td>$\sigma \to \text{Seq}(\sigma)$</td>
</tr>
<tr>
<td>$\text{isin}$</td>
<td>$\sigma \times \text{Seq}(\sigma) \to \text{Bool}$</td>
</tr>
<tr>
<td>$\text{cat}$</td>
<td>$\text{Seq}(\sigma) \times \text{Seq}(\sigma) \to \text{Seq}(\sigma)$</td>
</tr>
<tr>
<td>$\text{update}$</td>
<td>$\text{Seq}(\sigma) \times \text{Int} \times \sigma \to \text{Seq}(\sigma)$</td>
</tr>
<tr>
<td>$\text{index}$</td>
<td>$\text{Seq}(\sigma) \times \text{Int} \to \sigma$</td>
</tr>
<tr>
<td>$\text{range}$</td>
<td>$\text{Int} \times \text{Int} \to \text{Seq}(\text{Int})$</td>
</tr>
<tr>
<td>$\text{selection}$</td>
<td>$\text{Seq}(\sigma) \times \text{Seq}(\text{Int}) \to \text{Seq}(\sigma)$</td>
</tr>
<tr>
<td>$\text{restrict}$</td>
<td>$\text{Set}(\sigma) \times \text{Seq}(\sigma) \to \text{Seq}(\sigma)$</td>
</tr>
<tr>
<td>$\text{construct}$</td>
<td>$\text{Scope}(\sigma) \to \text{Seq}(\sigma)$</td>
</tr>
<tr>
<td>$\text{reduce}$</td>
<td>$\sigma \times \sigma \to \sigma \times \text{Seq}(\sigma) \to \sigma$</td>
</tr>
<tr>
<td>$\text{each}$</td>
<td>$\beta \to \sigma \times \text{Seq}(\beta) \to \text{Seq}(\sigma)$</td>
</tr>
<tr>
<td>$\text{subseq}$</td>
<td>$\text{Seq}(\sigma) \times \text{Seq}(\sigma) \to \text{Bool}$</td>
</tr>
<tr>
<td>$\text{prefix}$</td>
<td>$\text{Seq}(\sigma) \times \text{Seq}(\sigma) \to \text{Bool}$</td>
</tr>
<tr>
<td>$\text{length}$</td>
<td>$\text{Seq}(\sigma) \to \text{Int}$</td>
</tr>
</tbody>
</table>

stmt

$(\text{iterate} \; (\text{scope} \; ((\text{id} \; v) : \sigma \; (\text{isin} \; ((\text{id} \; v) : \sigma \; ((\text{exp} \; t) : \text{Seq}(\sigma)) \; (\text{stmt} \; S))))))$

$\sim \; \text{for} \; (v \; \parallel \; v \; \text{in} \; t \; \parallel \; S)$

props

See [Manna85].

Fig. 14. Type Definition $\text{Seq}(\sigma)$

with the usual restrictions regarding $t \in \text{def.} \; S$ and $i$ chosen so that $i \notin \text{fv.} \; S$. 
Bibliography

Aho74

Barringer84

Broy82

Burstall80

Burstall82

Burstall84

CIP83

Constable86
Dahl70

Demers84
A. Demers, J. Donahue, "Data Types are Values", Xerox PARC CSL report 83-5, 1984.

Dijkstra75

Dijkstra76

Emst82

Floyd67

Giarratana76

ADJ78

Gries78

Gries81

Gries85
Gries86

Guttag75

Guttag77

Guttag78

Hayes86

He86

Hehner86

Hoare69

Hoare72

Hoare72b

Hoare74
Hoare78

Hoare85

Hussman85

Jacobs85

Jones80

Jones86

Kamin83

Kamin83b

Kamin84

Lescanne83
Liskov75

Manna85

Milner71

Milner78

Morgan86

Nipkow85

Olderog84

Parnas71

Plotkin76

Prins86

Reynolds81
Robinson65

Sannella84

Scherlis86

Schoett85

Sherman83

Spivey86

Volpano86

Wand79

Wegbreit74

Wulf76