NOTE: This manual is in the process of being updated. Almost everything stated here is still correct, but incomplete due to the many new features that have been introduced into PVS over the years. The release notes should be consulted for the most current information.

The initial development of PVS was funded by SRI International. Subsequent enhancements were partially funded by SRI and by NASA Contracts NAS1-18969 and NAS1-20334, NRL Contract N00014-96-C-2106, NSF Grants CCR-9300044, CCR-9509931, and CCR-9712383, AFOSR contract F49620-95-C0044, and DARPA Orders E276, A721, D431, D855, and E301.
## Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Summary of the PVS Language</td>
<td>1</td>
</tr>
<tr>
<td>1.2 PVS Language Design Principles</td>
<td>2</td>
</tr>
<tr>
<td>1.3 An Example: stacks</td>
<td>4</td>
</tr>
<tr>
<td>2  The Lexical Structure</td>
<td>7</td>
</tr>
<tr>
<td>3  Declarations</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Type Declarations</td>
<td>12</td>
</tr>
<tr>
<td>3.1.1 Uninterpreted Type Declarations</td>
<td>12</td>
</tr>
<tr>
<td>3.1.2 Uninterpreted Subtype Declarations</td>
<td>13</td>
</tr>
<tr>
<td>3.1.3 Structural Subtypes</td>
<td>13</td>
</tr>
<tr>
<td>3.1.4 Empty and Singleton Record and Tuple Types</td>
<td>14</td>
</tr>
<tr>
<td>3.1.5 Interpreted Type Declarations</td>
<td>14</td>
</tr>
<tr>
<td>3.1.6 Enumeration Type Declarations</td>
<td>14</td>
</tr>
<tr>
<td>3.1.7 Empty versus Nonempty Types</td>
<td>15</td>
</tr>
<tr>
<td>3.1.8 Checking Nonemptiness</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Variable Declarations</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Constant Declarations</td>
<td>17</td>
</tr>
<tr>
<td>3.4 Recursive Definitions</td>
<td>18</td>
</tr>
<tr>
<td>3.5 Macros</td>
<td>21</td>
</tr>
<tr>
<td>3.6 Inductive and Coinductive Definitions</td>
<td>22</td>
</tr>
<tr>
<td>3.7 Formula Declarations</td>
<td>24</td>
</tr>
<tr>
<td>3.8 Judgements</td>
<td>25</td>
</tr>
<tr>
<td>3.8.1 Number and Name Judgements</td>
<td>26</td>
</tr>
<tr>
<td>3.8.2 Subtype Judgements</td>
<td>27</td>
</tr>
<tr>
<td>3.8.3 Judgement Processing</td>
<td>28</td>
</tr>
<tr>
<td>3.9 Conversions</td>
<td>28</td>
</tr>
<tr>
<td>3.9.1 Conversion Examples</td>
<td>29</td>
</tr>
<tr>
<td>3.9.2 Lambda conversions</td>
<td>30</td>
</tr>
<tr>
<td>CONTENTS</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
</tr>
</tbody>
</table>

3.9.3 Conversions on Type Constructors .......................... 31
3.9.4 Conversion Processing ........................................ 32
3.9.5 Conversion Control ........................................... 33
3.10 Library Declarations ........................................... 34
3.11 Auto-rewrite Declarations ............................... 34

4 Types ........................................................................ 41
4.1 Subtypes ................................................................. 41
4.2 Function Types .......................................................... 43
4.3 Tuple Types ............................................................... 44
4.4 Cotuple Types .......................................................... 45
4.5 Record Types ............................................................ 46
4.6 Dependent types ........................................................ 46
4.7 Record and Tuple Type Extensions ......................... 47

5 Expressions ................................................................ 49
5.1 Boolean Expressions .................................................. 50
5.2 IF-THEN-ELSE Expressions ........................................ 52
5.3 Numeric Expressions ................................................. 53
5.4 Characters and String Expressions ............................. 53
5.5 Applications .............................................................. 54
5.6 Binding Expressions .................................................. 54
5.7 LET and WHERE Expressions ..................................... 55
5.8 Set Expressions ........................................................ 56
5.9 Tuple Expressions ...................................................... 56
5.10 Projection Expressions ............................................... 56
5.11 Record Expressions .................................................. 57
5.12 Record Accessors ...................................................... 57
5.13 Cotuple Expressions .................................................. 58
5.14 Override Expressions ................................................. 59
5.15 Coercion Expressions ................................................ 60
5.16 Tables ................................................................. 60
   5.16.1 COND Expressions ........................................... 61
   5.16.2 Table Expressions ............................................. 63

6 Theories .................................................................. 67
6.1 Theory Identifiers ...................................................... 68
6.2 Theory Parameters .................................................... 69
6.3 Importings and Exportings ........................................ 69
   6.3.1 The EXPORTING Clause .................................. 69
   6.3.2 IMPORTING Clauses .......................................... 70
6.4 Assuming Part .......................................................... 71
6.5 Theory Part ............................................................... 73
7 Theory Interpretations  75

8 Name Resolution  77

9 Abstract Datatypes  81
  9.1 A Datatype Example: stack  82
  9.2 Datatype Details  86
  9.3 Datatype Subtypes  90
  9.4 CASES Expressions  91

A The Grammar  93

Bibliography  103

Index  105
Chapter 1

Introduction

PVS is a Prototype Verification System for the development and analysis of formal specifications. The PVS system primarily consists of a specification language, a parser, a type-checker, a prover, specification libraries, and various browsing tools. This document describes the specification language and is meant to be used as a reference manual. The PVS System Guide [11] is to be consulted for information on how to use the system to develop specifications and proofs. The PVS Prover Guide [16] is a reference manual for the commands used to construct proofs. The web site http://pvs.csl.sri.com provides many useful links, including various tutorials and examples.

In this section, we provide a brief summary of the PVS specification language, enumerate the key design principles behind the language, and discuss a simple stacks example.

1.1 Summary of the PVS Language

A PVS specification consists of a collection of theories. Each theory consists of a signature for the type names and constants introduced in the theory, and the axioms, definitions, and theorems associated with the signature. For example, a typical specification for a queue would introduce the queue type and the operations of enq, deq, and front with their associated types. In such a theory, one can either define a representation for the queue type and its associated operations in terms of some more primitive types and operations, or merely axiomatize their properties. A theory can build on other theories: for example, a theory for ordered binary trees could build on the theory for binary trees. A theory can be parametric in certain specified types and values: as examples, a theory of queues can be parametric in the maximum queue length, and a theory of ordered binary trees can be parametric in the element type as well as the ordering relation. It is possible to place constraints, called assumptions, on the parameters of a theory so that, for instance, the ordering relation parameter of an ordered binary tree can be constrained to be a total ordering.

The PVS specification language is based on simply typed higher-order logic. Within a theory, types can be defined starting from base types (Booleans, numbers, etc.) using type constructors such as function, record, and tuple types. The terms of the language can
be constructed using, for example, function application, lambda abstraction, and record or
tuple constructions.

There are a few significant enhancements to the simply typed language above that lend
considerable power and sophistication to PVS. New uninterpreted base types may be intro-
duced. One can define a predicate subtype of a given type as the subset of individuals in a
type satisfying a given predicate: the subtype of nonzero reals is written as \( \{ x: \text{real} \mid x \\neq 0 \} \). One benefit of such subtyping is that when an operation is not defined on all the el-
ements of a type, the signature can directly reflect this. For example, the division operation
on reals is given a type where the denominator is constrained to be nonzero. Typechecking
then ensures that division is never applied to a zero denominator. Since the predicate used
in defining a predicate subtype is arbitrary, typechecking is undecidable and may lead to
proof obligations called type correctness conditions (TCCs). The user is expected to dis-
charge these proof obligations with the assistance of the PVS prover. The PVS type system
also features dependent function, record, and tuple type constructions. There is also a fa-
cility for defining a certain class of abstract datatype (namely well-founded trees) theories
automatically.

1.2 PVS Language Design Principles

There are several basic principles that have motivated the design of PVS which are expli-
cated in this section.

**Specification vs. Programming Languages.** A specification represents requirements or
a design whereas a program text represents an implementation of a design. A program can
be seen as a specification, but a specification need not be a program. Typically, a specifi-
cation expresses what is being computed whereas a program expresses how it is computed.
A specification can be incomplete and still be meaningful whereas an incomplete program
will typically not be executable. A specification need not be executable; it may use high-
level constructs, quantifiers and the like, that need have no computational meaning. How-
ever, there are a number of aspects of programming languages that a specification language
should include, such as:

- the usual basic types: booleans, integers, and rational numbers
- the familiar datatypes of programming languages such as arrays, records, lists, se-
quences, and abstract datatypes
- the higher-order capabilities provided by modern functional programming languages
  so that extremely general-purpose operations can be defined
- definition by recursion
- support for dividing large specifications into parameterized modules
It is clearly not enough to say that a specification language shares some important features of a programming language but need not be executable. Any useful formal language must have a clearly defined semantics and must be capable of being manipulated in ways that are meaningful relative to the semantics. A programming language for example can be given a denotational semantics so that the execution of the program respects its denotational meaning. The reason one writes a specification in a formal language is typically to ensure that it is sensible, to derive some useful consequences from it, and to demonstrate that one specification implements another. All of these activities require the notion of a justification or a proof based on the specification, a notion that can only be captured meaningfully within the framework of logic.

Untyped set theory versus higher-order logic Which logic should be chosen? There is a wide variety of choices: simple propositional logics, which can be classical or intuitionistic, equational logics, quantificational logics, modal and temporal logics, set theory, higher-order logic, etc. Some propositional and modal logics are appropriate for dealing with finite state machines where one is primarily interested in efficiently deciding certain finite state machine properties. For a general purpose specification language, however, only a set theory or a higher-order logic would provide the needed expressiveness. Higher-order logic requires strict typing to avoid inconsistencies whereas set theory restricts the rules for forming sets. Set theory is inherently untyped, and grafting a typechecker onto a language based on set theory is likely to be too strict and arbitrary. Typechecking, however, is an extremely important and easy way of checking whether a specification makes semantic sense (although for an opposing view, the reader is referred to a report by Lamport and Paulson [9]). Higher-order logic does admit effective typechecking but at the expense of an inflexible type system. Recent advances in type theory have made it possible to design more flexible type systems for higher-order logic without losing the benefits of typechecking. We have therefore chosen to base PVS on higher-order logic.

Total versus partial functions In the PVS higher-order logic, an individual is either a function, a tuple, a record, or the member of a base type. Functions are extremely important in higher-order logic. They are first-class individuals, i.e., variables can range over functions. In general, functions can represent either total or partial maps. A total map from domain $A$ to range $B$ maps each element of $A$ to some element of $B$, whereas a partial map only maps some of the elements of $A$ to elements of $B$. Most traditional logics build in the assumption that functions represent total maps. Partial functions arise quite naturally in specifications. For example, the division operation is undefined on a zero denominator and the operation of popping a stack is undefined on an empty stack.

Some recent logics, notably those of VDM [8], LUTINS [5], RAISE [6], Beeson [2] and Scott [15], admit partial functions. In these logics, some terms may be undefined by not denoting any individuals. Some of these logics have mechanisms for distinguishing defined
and undefined terms, while others allow “undefined” to propagate from terms to expressions and therefore must employ multiple truth values. In all these cases, the ability to formalize partially defined functions comes at the cost of complicating the deductive apparatus, even when the specification does not involve any partial functions. Though logics that allow partial functions are extremely interesting, we have chosen to avoid partial functions in PVS. We have instead employed the notion of a predicate subtype, a type that consists of those elements of a given type satisfying a given predicate. Using predicate subtypes, the type of the division operator, for example, can be constrained to admit only nonzero denominators. Division then becomes a total operation on the domain consisting of arbitrary numerators and nonzero denominators. The domain of a pop operation on stacks can be similarly restricted to nonempty stacks. PVS thus admits partial functions within the framework of a logic of total functions by enriching the type system to include predicate subtypes. We find this use of predicate subtypes to be significantly in tune with conventional mathematical practice of being explicit about the domain over which a function is defined.

1.3 An Example: stacks

In this section we discuss a specific example, the theory of stacks, in order to give a feel for the various aspects of the PVS language before going into detail. Apart from the basic notation for defining a theory, this example illustrates the use of type parameters at the theory level, the general format of declarations, the use of predicate subtyping to define the type of nonempty stacks, and the generation of typechecking obligations.

```
stacks [t: TYPE+] : THEORY
BEGIN
  stack : TYPE+
  s : VAR stack
  empty : stack
  nonemptystack?(s) : bool = s /= empty
  push : [t, stack -> (nonemptystack?)]
  pop : [(nonemptystack?) -> stack]
  top : [(nonemptystack?) -> t]
  x, y : VAR t
  push_top_pop : AXIOM
    nonemptystack?(s) IMPLIES push(top(s), pop(s)) = s
  pop_push : AXIOM pop(push(x, s)) = s
  top_push : AXIOM top(push(x, s)) = x
  pop2push2 : THEOREM pop(pop(push(x, push(y, s)))) = s
END stacks
```

Figure 1.1: Theory stacks

Figure 1.1 illustrates a theory for stacks of an arbitrary type with corresponding stack
operations. Note that this is not the recommended approach to specifying stacks; a more convenient and complete specification is provided in Section 9.1, page 82.

The first line introduces a theory named stacks that is parameterized by a type t (the formal parameter of stacks). The keyword TYPE+ indicates that t is a non-empty type. The uninterpreted (nonempty) type stack is declared, and the constant empty and variable s are declared to be of type stack. The defined predicate nonemptystack? is then declared on elements of type stack; it is true for a given stack element iff that element is not equal to empty.3 The functions push, pop, and top are then declared. Note that the predicate nonemptystack? is being used as a type in specifying the signatures of these functions; any predicate may be used where a type is expected simply by putting parentheses around it.

The variables x and y are then declared, followed by the usual axioms for push, pop, and top, which make push a stack constructor and pop and top stack accessors. Finally, there is the theorem pop2push2, that can easily be proved by two applications of the pop_push axiom.

This simple theorem has an additional facet that shows up during typechecking. Note that pop expects an element of type (nonemptystack?) and returns a value of type stack. This works fine for the inner pop because it is applied to push, which returns an element of type (nonemptystack?): but the outer occurrence of pop cannot be seen to be type correct by such syntactic means. In cases like these, where a subtype is expected but not directly provided, the system generates a type-correctness condition (TCC). In this case, the TCC is pop2push2_TCC1: OBLIGATION
FORALL (s: stack, x, y: t): nonemptystack?(pop(push(x, push(y, s))));
and is easily proved using the pop_push axiom. The system keeps track of all such obligations and will flag the unproved ones during proof chain analysis.

Parameterized theories such as stacks introduce theory schemas, where the type t may be instantiated with any other nonempty type. To use the types, constants, and formulas of the stacks theory from another theory, the stacks theory must be imported, with actual parameters provided for the corresponding theory parameters. This is done by means of an IMPORTING clause. For example, consider the theory ustacks.

```
ustacks : THEORY
BEGIN
IMPORTING stacks[int], stacks[stack[int]]

si : stack[int]
sos : stack[stack[int]] = push(si, empty)
END ustacks
```

It imports stacks of integers and stacks of stacks of integers. The constant si is then declared to be a stack of integers, and the constant sos is a stack of stacks of integers whose top element is si. Note that the system is able to determine which instance of push and empty is meant from the type of the first argument. In general, the typechecker infers the type of an expression from its context.

---

2If and only if.
3The bool type and /= operator are declared in the prelude, which is a large body of theories that are preloaded into PVS.
Introduction

1.3 An Example: stacks

The following chapters provide more details on the various features of the language. The lexical aspects of the language are explained in Chapter 2. Chapter 3 describes declarations, Chapters 4 and 5 describe type expressions and expressions, and Chapter 6 explains theories, theory parameters, and the importing and exporting of names. Theory interpretations and mappings are described in Chapter 7. Chapter 8 describes names and name resolution, and Chapter 9 details the datatype facility of PVS. Finally, Appendix A provides the grammar of the language.
Chapter 2

The Lexical Structure

PVS specifications are text files, each composed of a sequence of lexical elements which in turn are made up of characters. The lexical elements of PVS are the identifiers, reserved words, special symbols, numbers, whitespace characters, and comments.

Identifiers are composed of letters, digits, and the characters \_ or ?; they must begin with a letter, which are the usual ASCII letters, or any Unicode character that is not one of the ASCII non-letter characters. Note that keywords from Figure 2.2 and operators from Figure 2.3 may be embedded in identifiers, but may not be identifiers themselves. Thus candy is an identifier, though it contains the keyword and, and ∧∧ is an identifier, though it contains the keyword ∧.

They may be arbitrarily long. Identifiers are case-sensitive; F00, Foo, and foo are different identifiers. PVS strings contain any Unicode character: to include a " in the string, use \" and to include a \ use \\n. For more on Unicode, see the PVS User Guide.

```
|  Ids  | ::= | Id^+ |
|  Id   | ::= | Letter IdChar^+ |
| Number| ::= | Digit^+ |
| String| ::= | " Unicode-character* " |
| IdChar| ::= | Letter | Digit | _ | ? |
| Letter| ::= | A | ... | Z | a | ... | z | Non-ASCII-Unicode-character |
| Digit | ::= | 0 | ... | 9 |
```

Figure 2.1: Lexical Syntax

The reserved words are shown in Figure 2.2. Unlike identifiers, they are not case-sensitive. In this document, reserved words are always displayed in upper case. Note that identifiers may have reserved words embedded in them, thus ARRAYALL is a valid identifier and will not be confused with the two embedded reserved words. The meaning of the
reserved words are given in the appropriate sections; they are collected here for reference.

The special symbols are listed in Figure 2.3. All of these symbols are separators; they separate identifiers, numbers, and reserved words.

The whitespace characters are space, tab, newline, return, and newline; they are used to separate other lexical elements. At least one whitespace character must separate adjacent identifiers, numbers, and reserved words.

Comments may appear anywhere that a whitespace character is allowed. They consist of the ‘%’ character followed by any sequence of characters and terminated by a newline.

The definable symbols are shown in Table 2.4. These keywords and symbols may be declared if declared any number of times, though this may lead to ambiguities. Such ambiguities may be resolved by including the theory name, actual parameters, and possibly the type as a coercion.

Symbols that are binary infix (Binop), for example AND and +, may be declared with any number of arguments. If they are declared with two arguments then they may subsequently be used in prefix or infix form. Otherwise they may only be used in prefix form. Similarly for unary operators, and the IF operator, which may be used in IF-THEN-ELSE-ENDIF form if declared with three arguments.

Note that when typing the operators / or outside of a specification, the backslash may need to be doubled (or in rare cases, quadrupled). This is because it is commonly used as an “escape” character, and the character following may be interpreted specially.

The symbol pairs [ ] | |, ( | |), { | |}, ⟨ ⟩, ⟦ ⟧, ⌈ ⌉, ⌊ ⌋, ⌜ ⌝, and ⟯ are available

1In particular, & * + - / = . = , , << <<= = <<= > > = = AND IFF IMPLIES NOT 0 OR WHEN XOR ^ and ~ are declared there. Note that many of these are overloaded, for example, ^ has three different definitions.
as outfix operators. They are declared and may be used by concatenation, for example, with the declaration \[\text{[:]}: \text{[bool, int -> int]}\] the outfix term \[\text{:} \text{TRUE, 0 \text{[:]} \text{}}\] is equivalent to the prefix form \[\text{[:]} \text{([[}\text{}} \text{TRUE, 0 \text{[} \text{]}\text{)}}\].
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td># #</td>
<td>AND</td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp; AND THEN</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>IF</td>
</tr>
<tr>
<td>**</td>
<td>IFF</td>
</tr>
<tr>
<td>+</td>
<td>IMPLIES</td>
</tr>
<tr>
<td>++</td>
<td>NOT</td>
</tr>
<tr>
<td>-</td>
<td>O</td>
</tr>
<tr>
<td>/</td>
<td>OR</td>
</tr>
<tr>
<td>//</td>
<td>ORELSSE</td>
</tr>
<tr>
<td>/ =</td>
<td>TRUE</td>
</tr>
<tr>
<td>\ \</td>
<td>WHEN</td>
</tr>
<tr>
<td>&lt;</td>
<td>XOR</td>
</tr>
<tr>
<td>&lt;=</td>
<td>[ ]</td>
</tr>
<tr>
<td>&lt;=</td>
<td>[</td>
</tr>
<tr>
<td>&lt;=</td>
<td>\</td>
</tr>
<tr>
<td>&lt;=</td>
<td>^</td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;&lt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>=&gt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&gt;&gt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>@@</td>
</tr>
<tr>
<td>@ @</td>
<td>OR ELSE</td>
</tr>
</tbody>
</table>

Figure 2.4: PVS Definable Symbols
Chapter 3

Declarations

Entities of PVS are introduced by means of declarations, which are the main constituents of PVS specifications. Declarations are used to introduce types, variables, constants, formulas, judgements, conversions, and other entities. Most declarations have an identifier and belong to a unique theory. Declarations also have a body which indicates the kind of the declaration and may provide a signature or definition for the entity. Top-level declarations occur in the formal parameters, the assertion section and the body of a theory. Local declarations for variables may be given, in association with constant and recursive declarations and binding expressions (e.g., involving FORALL or LAMBDA). Declarations are ordered within a theory; earlier declarations may not reference later ones.\(^1\)

Declarations introduced in one theory may be referenced in another by means of the IMPORTING and EXPORTING clauses. The EXPORTING clause of a theory indicates those entities that may be referenced from outside the theory. There is only one such clause for a given theory. The IMPORTING clauses provide access to the entities exported by another theory. There can be many IMPORTING clauses in a theory; in general they may appear anywhere a top-level declaration is allowed. See Section 6.3 for more details.

PVS allows the overloading of declaration identifiers. Thus a theory named foo may declare a constant foo and a formula foo. To support this ad hoc overloading, declarations are classified according to kind; in PVS the primary kinds are type, prop, expr, and theory. Type declarations are of kind type, and may be referenced in type declarations, actual parameters, signatures, and expressions. Formula declarations are of kind prop, and may be referenced in auto-rewrite declarations (Section 3.11) or proofs (see the PVS Prover Guide [16]). Variable, constant, and recursive definition declarations are of kind expr; these may be referenced in expressions and actual parameters. Newly introduced names need only be unique within a kind, as there is no way, for example, to use an expression where a type is expected.\(^2\)

\(^{1}\)Thus mutual recursion is not directly supported. The effect can be achieved with a single recursive function that has an argument that serves as a switch for selecting between two or more subexpressions.

\(^{2}\)There are a few exceptions, for example the actual parameters of theories, since theories may be instantiated with types or expressions.
Declarations generally consist of an identifier, an optional list of bindings, and a body. The body determines the kind of the declaration, and the bindings and the body together determine the signature and definition of the declared entity. Multiple declarations may be given in compressed form in which a common body is specified for multiple identifiers; for example

\[
\begin{align*}
  x, y, z & \colon \text{VAR int} \\
\end{align*}
\]

In every case this is treated the same as the expanded form, thus the above is equivalent to:

\[
\begin{align*}
  x & \colon \text{VAR int} \\
  y & \colon \text{VAR int} \\
  z & \colon \text{VAR int} \\
\end{align*}
\]

In the rest of this chapter we describe declarations for types, variables, constants, recursive definitions, macros, inductive and coinductive definitions, formulas, judgements, conversions, libraries, and auto-rewrites. The declarations for theory parameters, importings, exportings, and theory abbreviations are given in Chapter 6. Figure 3.1 gives the syntax for declarations.

3.1 Type Declarations

Type declarations are used to introduce new type names to the context. There are four kinds of type declaration:

- uninterpreted type declaration: \( T : \text{TYPE} \)
- uninterpreted subtype declaration: \( S : \text{TYPE FROM } T \)
- interpreted type declaration: \( T : \text{TYPE} = \text{int} \)
- enumeration type declarations: \( T : \text{TYPE} = \{ r, g, b \} \)

These type declarations introduce type names that may be referenced in type expressions (see Section 4). They are introduced using one of the keywords \text{TYPE}, \text{NONEMPTY_TYPE}, or \text{TYPE+}.

3.1.1 Uninterpreted Type Declarations

Uninterpreted types support abstraction by providing a means of introducing a type with a minimum of assumptions on the type. An uninterpreted type imposes almost no constraints on an implementation of the specification. The only assumption made on an uninterpreted type \( T \) is that it is disjoint from all other types, except for subtypes of \( T \). For example,

\[
\begin{align*}
  T_1, T_2, T_3 & \colon \text{TYPE} \\
\end{align*}
\]

introduces three new pairwise disjoint types. If desired, further constraints may be put on these types by means of axioms or assumptions (see Section 3.7 on page 24).

It should be emphasized that uninterpreted types are important in providing the right level of abstraction in a specification. Specifying the type body may have the undesired effect of restricting the possible implementations, and cluttering the specification with needless detail.
3.1.2 Uninterpreted Subtype Declarations

Uninterpreted subtype declarations are of the form

\[ s: \text{TYPE FROM} \ t \]

This introduces an uninterpreted subtype \( s \) of the supertype \( t \). This has the same meaning as

\[ s: \text{TYPE} = (s\_\text{pred}) \]

in which a new predicate is introduced in the first line and the type \( s \) is declared as a predicate subtype in the second line\(^3\). No assumptions are made about uninterpreted subtypes; in particular, they may or may not be empty, and two different uninterpreted subtypes of the same supertype may or may not be disjoint. Of course, if the supertypes themselves are disjoint, then the uninterpreted subtypes are as well.

3.1.3 Structural Subtypes

PVS has support for structural subtyping for record and tuple types. A record type \( S \) is a structural subtype of record type \( R \) if every field of \( R \) occurs in \( S \), and similarly, a tuple type \( T \) is a structural subtype of a tuple type forming a prefix of \( T \). Section 4.7 gives examples, as \( \text{colored_point} \) is a structural subtype of \( \text{point} \), and \( R_5 \) is a structural subtype of \( R_3 \). Structural subtypes are akin to the class hierarchy of object-oriented systems, where the fields of a record can be viewed as the slots of a class instance. The PVS equivalent of setting a slot value is the override expression (sometimes called update), and this works with structural subtypes, allowing the equivalent of generic methods to be defined. Here is an example:

```pvs
points: THEORY
BEGIN
  point: \text{TYPE}+ = [# x, y: \text{real} #]
END points

genpoints[(IMPORTING points) gpoint: \text{TYPE} <: point]: THEORY
BEGIN
  move(p: gpoint)(dx, dy: \text{real}): gpoint =
  \quad p \text{ WITH } ['x := p'x + dx, 'y := p'y + dy]
END genpoints

colored_points: THEORY
BEGIN
  IMPORTING points
  Color: \text{TYPE} = \text{red, green, blue}
  colored_point: \text{TYPE} = \text{point WITH } [# color: Color #]
  IMPORTING genpoints[colored_point]
  p: colored_point
  move0: LEMMA move(p)(0, 0) = p
END colored_points
```

The declaration for \( gpoint \) uses the structural subtype operator \(<: \). This is analogous to the FROM keyword, which introduces a (predicate) subtype. This example also serves to explain why we chose to separate structural and predicate subtyping. If they were treated

\(^3\)This is described in Section 4.1 (page 41).
uniformly, then \texttt{gpoint} could be instantiated with the unit disk; but in that case the \texttt{move} operator would not necessarily return a \texttt{gpoint}. The TCC could not be generated for the \texttt{move} declaration, but would have to be generated when the \texttt{move} was referenced. This both complicates typechecking, and makes TCCs and error messages more inscrutable. If both are desired, simply include a structural subtype followed by a predicate subtype, for example:

\begin{verbatim}
genpoints[(IMPORTING points) gpoint: TYPE <: point, spoint: TYPE FROM gpoint]: THEORY
\end{verbatim}

Now \texttt{move} may be applied to \texttt{gpoints}, but if applied to a \texttt{spoint} an unprovable TCC will result.

Structural subtypes are a work in progress. In particular, structural subtyping could be extended to function and datatypes. And to have real object-oriented PVS, we must be able to support a form of method invocation.

### 3.1.4 Empty and Singleton Record and Tuple Types

Empty and singleton record and tuple types are now allowed in PVS. Thus the following are valid declarations:

\begin{verbatim}
Tup0: TYPE = [ ]
Tup1: TYPE = [int]
Rec0: TYPE = [# #]
\end{verbatim}

Note that the space is important in the empty tuple type, as otherwise it is taken to be an operator (the box operator).

### 3.1.5 Interpreted Type Declarations

Interpreted type declarations are primarily a means for providing names for type expressions. For example,

\begin{verbatim}
intfun: TYPE = [int -> int]
\end{verbatim}

introduces the type name \texttt{intfun} as an abbreviation for the type of functions with integer domain and range. Because PVS uses \textit{structural equivalence} instead of \textit{name equivalence}, any type expression \texttt{T} involving \texttt{intfun} is equivalent to the type expression obtained by substituting \texttt{[int -> int]} for \texttt{intfun} in \texttt{T}. The available type expressions are described in Chapter 4 on page 41.

Interpreted type declarations may be given parameters. For example, the type of integer subranges may be given as

\begin{verbatim}
subrange(m, n: int): TYPE = {i:int | m <= i AND i <= n}
\end{verbatim}

and \texttt{subrange} with two integer parameters may subsequently be used wherever a type is expected. Any use of a parameterized type must include all of the parameters, so currying of the parameters is not allowed. Note that \texttt{subrange} may be overloaded to declare a different type in the same theory without any ambiguity, as long as the number or type of parameters is different.

### 3.1.6 Enumeration Type Declarations

Enumeration type declarations are of the form
3.1 Type Declarations

**enum: TYPE = \{e_1, \ldots, e_n\}**

where the e_i are distinct identifiers which are taken to completely enumerate the type. This is actually a shorthand for the datatype specification

```
enum: DATATYPE
e_1: e_1?
\vdots
e_n: e_n?
END enum
```

explained in Chapter 9. Because of this, enumeration types may only be given as top-level declarations, and are not type expressions. The advantage of treating them as datatypes is that the necessary axioms are automatically generated, and the prover has built-in facilities for handling datatypes.

3.1.7 Empty versus Nonempty Types

As noted before, PVS allows empty types, and the term *type* refers to either empty or nonempty types. Constants declared to be of a given type provide elements of the type, so the type must be nonempty or there is an inconsistency. Thus whenever a constant is declared, the system checks whether the type is nonempty, and if it cannot decide that it is nonempty it generates an *existence TCC*. This is the simple explanation, but it is made somewhat complicated by the considerations of formal parameters, uninterpreted versus interpreted type declarations, explicit declarations of nonemptiness, and *CONTAINING* clauses on type declarations, as well as a desire to keep the number of TCCs generated to a minimum, while guaranteeing soundness. The details are provided below.

First note that having variables range over an empty type causes no difficulties, so variable declarations and variable bindings never trigger the nonemptiness check.

During typechecking, type declarations may indicate that the type is nonempty, and constant declarations of a given type require that the type be nonempty. When a type is determined to be nonempty, it is marked as such so that future checks of constants do not trigger more TCCs. Below we describe how type declarations are handled first for declarations in the body of a theory, and then for type declarations that appear in the formal parameters, as they require special handling.

**Theory Body Type Declarations**

- Uninterpreted type or subtype declarations introduced with the keyword TYPE may be empty. Declaring a constant of that type will lead to a TCC that is unprovable without further axioms.

- Uninterpreted type declarations introduced with the keyword NONEMPTY_TYPE or TYPE+ are assumed to be nonempty. Thus the type is marked nonempty.

---

4If the type T is empty, then the following two equivalences hold:

\[(\forall x: T): p(x) \iff \text{TRUE} \quad \text{and} \quad (\exists x: T): p(x) \iff \text{FALSE}\]
• Uninterpreted subtype declarations introduced with the keyword _NONEMPTY_TYPE_ or _TYPE+_ are assumed to be nonempty, as long as the supertype is nonempty. Thus the supertype is checked, and an existence TCC is generated if the supertype is not known to be nonempty. Then the subtype is marked nonempty.

• The type of an interpreted constant is nonempty, as the definition provides a witness.

• Interpreted type declarations introduced with the keyword _TYPE_ may be nonempty, depending on the type definition.

• Any interpreted type declaration with a _CONTAINING_ clause is marked nonempty, and the _CONTAINING_ expression is typechecked against the specified type. In this case no existence TCC is generated, since the _CONTAINING_ expression is a witness to the type. Of course, other TCCs may be generated as a result of typechecking the _CONTAINING_ expression.

Formal Type Declarations  Only uninterpreted (sub)type declarations may appear in the formal parameters list.

• Formal type declarations introduced with the _TYPE_ keyword may be empty. This is handled according to the occurrences of constant declarations involving the type.

• If there is a constant declaration of that type in the formal parameter list, then no TCCs are generated, since any instance of the theory will need to provide both the type and a witness. The type is marked nonempty in this case.

• If the type declaration is a formal parameter and a constant is declared whose type involves the type, but is not the type itself (for example, if the formal theory parameters are \( \{ t: \_TYPE_, \ f: [t \rightarrow t] \} \)), then a TCC may be generated, and a comment is added to the TCC indicating that an assuming clause may be needed in order to discharge the TCC. This TCC will be generated only if an earlier constant declaration hasn’t already forced the type to be marked nonempty. Note that there are circumstances in which the formal type may be empty but the type expression involving that type is nonempty. This is discussed further below.

### 3.1.8 Checking Nonemptiness

The typechecker knows a type to be nonempty under the following circumstances:

• The type was declared to be nonempty, using either the _NONEMPTY_TYPE_ or the synonymous _TYPE+_ keyword. If the type is uninterpreted, this amounts to an assumption that the type is nonempty. If the type has a definition, then an existence TCC is generated unless the defining type expression is known to be nonempty.

• The type was declared to have an element using a _CONTAINING_ expression.
• A constant was declared for the type. In this case an existence TCC is generated for the first such constant, after which the type is marked as nonempty.

• It was marked as nonempty from an earlier check.

Once an unmarked type is determined to be nonempty, it is marked by the typechecker so that later checks will not generate existence TCCs. In addition, the type components are marked as nonempty. Thus the types that make up a tuple type, the field types of a record type, and the supertype of a subtype are all marked.

It is possible for two equivalent types to be marked differently, for example:

\[ t_1: \text{TYPE} = \{x: \text{int} \mid x > 2\} \]
\[ t_2: \text{TYPE} = \{x: \text{int} \mid x > 2\} \]
\[ c_1: t_1 \]

only marks the first type (\(t_1\)). Hence, it is best to name your types and to use those names uniformly.

### 3.2 Variable Declarations

Variable declarations introduce new variables and associate a type with them. These are logical variables, not program variables; they have nothing to do with state—they simply provide a name and associated type so that binding expressions and formulas can be succinct. Variables may not be exported. Variable declarations also appear in binding expressions such as \textsc{forall} and \textsc{lambda}. Such local declarations “shadow” any earlier declarations. For example, in

\[ x: \text{VAR} \text{bool} \]
\[ f: \text{FORMULA} (\text{forall} (x: \text{int})$: (\exists x: \text{nat}) p(x)) \text{ AND } q(x)) \]

The occurrence of \(x\) as an argument to \(p\) is of type \text{nat}, shadowing the one of type \text{int}. Similarly, the occurrence of \(x\) as an argument to \(q\) is of type \text{int}, shadowing the one of type \text{bool}.

### 3.3 Constant Declarations

Constant declarations introduce new constants, specifying their type and optionally providing a value. Since PVS is a higher order logic, the term \textit{constant} refers to functions and relations, as well as the usual (0-ary) constants. As with types, there are both uninterpreted and interpreted constants. Uninterpreted constants make no assumptions, although they require that the type be nonempty (see Section 3.1.8, page 16). Here are some examples of constant declarations:

\[ n: \text{int} \]
\[ c: \text{int} = 3 \]
\[ f: [\text{int} \rightarrow \text{int}] = (\lambda x: \text{int}: x + 1) \]
\[ g(x: \text{int}): \text{int} = x + 1 \]

The declaration for \(n\) simply introduces a new integer constant. Nothing is known about this constant other than its type, unless further properties are provided by \textsc{axiom}s. The other three constants are interpreted. Each is equivalent to specifying two declarations: \textit{e.g.}, the third line is equivalent to
3.4 Recursive Definitions

Recursive definitions are treated as constant declarations, except that the defining expression is required, and a measure must be provided, along with an optional well-founded order relation. The same syntax for arguments is available as for constant declarations; see the preceding section.

PVS allows a restricted form of recursive definition; mutual recursion is not allowed, and the function must be total, so that the function is defined for every value of its domain.

3.4 Recursive Definitions

Recursive definitions are treated as constant declarations, except that the defining expression is required, and a measure must be provided, along with an optional well-founded order relation. The same syntax for arguments is available as for constant declarations; see the preceding section.

PVS allows a restricted form of recursive definition; mutual recursion is not allowed, and the function must be total, so that the function is defined for every value of its domain.
In order to ensure this, recursive functions must be specified with a *measure*, which is a function whose signature matches that of the recursive function, but with range type the domain of the order relation, which defaults to $<$ on nat or ordinal. If the order relation is provided, then it must be a binary relation on the range type of the measure, and it must be well-founded; a *well-founded TCC* is generated if the order is not declared to be well-founded.

Here is the classic example of the factorial function:

\[
\text{factorial}(x: \text{nat}): \text{RECURSIVE nat} = \begin{cases} 
\text{IF } x = 0 \text{ THEN } 1 \text{ ELSE } x \ast \text{factorial}(x - 1) \text{ ENDIF} 
\end{cases}
\]

The measure is the expression following the MEASURE keyword (the optional order relation follows a BY keyword after the measure). This definition generates a *termination TCC*; a proof obligation which must be discharged in order that the function be well-defined. In this case the obligation is

\[
\text{factorial}\_TCC2: \text{OBLIGATION} \\
\text{FORALL } (x: \text{nat}): \text{NOT } x = 0 \implies x - 1 < x
\]

It is possible to abbreviate the given MEASURE function by leaving out the LAMBDA binding. For example, the measure function of the factorial definition may be abbreviated to:

\[
\text{MEASURE } x
\]

The typechecker will automatically insert a lambda binding corresponding to the arguments to the recursive function if the measure is not already of the correct type, and will generate a typecheck error if this process cannot determine an appropriate function from what has been specified.

A termination TCC is generated for each recursive occurrence of the defined entity within the body of the definition.\(^5\) It is obtained in one of two ways. If a given recursive reference has at least as many arguments provided as needed by the measure, then the TCC is generated by applying the measure to the arguments of the recursive call and comparing that to the measure applied to the original arguments using the order relation. The factorial TCC is of this form. The context of the occurrence is included in the TCC; in this case the occurrence is within the ELSE part of an IF-THEN-ELSE so the negated condition is an antecedent to the proof obligation.

If the reference does not have enough arguments available, then the reference is actually given a *recursive signature* derived from the recursive function as described below. This type constrains the domain to satisfy the measure, and the termination TCC is generated as a *termination-subtype TCC*. Termination-subtype TCCs are recognized as such by the occurrence of the order in the goal of the TCC. For example, we could define a substitution function for terms as follows.

\[^5\text{Some of these may be subsumed by earlier TCCs, and hence will not be displayed with the } M-x \text{ show-tccs command.}\]
### 3.4 Recursive Definitions

#### Declarations

```plaintext
term: DATATYPE
BEGIN
    mk_var(index: nat): var?
    mk_const(index: nat): const?
    mk_apply(fun: term, args: list[term]): apply?
END term

subst(x: (var?), y: term)(s: term): RECURSIVE term =
    (CASES s OF
        mk_var(i): (IF index(x) = i THEN y ELSE s ENDIF),
        mk_const(i): s,
        mk_apply(t, ss): mk_apply(subst(x, y)(t), map(subst(x, y))(ss))
    ENDCASES)
MEASURE s BY <<
```

Now the first recursive occurrence of `subst` has all arguments provided, so the termination TCC is as expected. The second occurrence does not have enough arguments. The recursive signature of that occurrence is

\[ \{z_1: \text{term} \mid z_1 << s\} \rightarrow \text{term} \]

Hence the signature of `subst(x, y)` is \[ \{z_1: \text{term} \mid z_1 << s\} \rightarrow \text{term} \], and map is instantiated to `map[\{z_1: \text{term} \mid z_1 << s\}, \text{term}]`, which leads to the TCC

```plaintext
subst_TCC2: OBLIGATION
    FORALL (ss: list[term], t: term, s: term, x: (var?):
        s = mk_apply(t, ss) IMPLIES every[term](LAMBDA (z: term): z << s)(ss);
```

Note that this map instance could be given directly, just don’t make the mistake of providing `map[\text{term, term}]`, as this leads to a TCC that says every `\text{term}` is \( << \text{s} \). For the same reason, if the uncurried form of this definition is given, then a lambda expression will have to be provided and the type will have to include the measure, for example,

```plaintext
subst(x: (var?), y, s: term): RECURSIVE term =
    (CASES s OF
        mk_var(i): (IF index(x) = i THEN y ELSE s ENDIF),
        mk_const(i): s,
        mk_apply(t, ss): mk_apply(subst(x, y, t),
            map(LAMBDA (s1: \{z: \text{term} \mid z << s\}): subst(x, y, s1))(ss))
    ENDCASES)
MEASURE s BY <<
```

The recursive signature is generated based on the type of the recursive function and the measure. For curried functions, it may be that the measure does not have the entire domain of the recursive function, but only the first few. For example, consider the measure for the function `f`.

```plaintext
f(r: real)(x, y: nat)(b: boolean): RECURSIVE boolean
    = ...
```

The type of the measure function is \[ \{\text{real} \rightarrow \{\text{nat, nat} \rightarrow \text{nat}\}\} \], which is a prefix of the function type. In deriving the recursive signature, the last domain type of the measure is constrained (using a subtype) in the corresponding position of the recursive function type. In this case the recursive signature is

\[ \{\text{real} \rightarrow \{\text{nat} \rightarrow \text{nat}\}\} \rightarrow \text{boolean} \]
Macros

There are some definitions that are convenient to use, but it’s preferable to have them expanded whenever they are referenced. To some extent this can be accomplished using auto-rewrites in the prover, but rewriting is restricted. In particular terms in types or actual

\[ \text{real} \rightarrow \{z: \text{nat}, \text{nat} | z'1 < x \} \rightarrow \{\text{boolean} \rightarrow \text{boolean}\} \]

Note that the recursive signature is a dependent type that depends on the arguments of the recursive function (x in this case), and hence only applies within the body of the recursive definition.

The formal argument that typechecking the body of a recursive function using the recursive signature is sound will appear in a future version of the semantics manual, for now note that simple attempts to subvert this mechanism do not work, as the following example illustrates.

\[
\text{fbad: RECURSIVE } [\text{nat} \rightarrow \text{nat}] = \text{fbad}
\]

\[
\text{MEASURE lambda } (n: \text{nat}): n
\]

This leads an unprovable TCC.

\[
\text{fbad TCC1: OBLIGATION FORALL } (x1: \text{nat}, x: \text{nat}): x < x1;
\]

The TCC results from the comparison of the expected type \([\text{nat} \rightarrow \text{nat}]\) to the derived type \([\{z: \text{nat} | z < x1\} \rightarrow \text{nat}\] . Remember that in PVS domains of function types must be equal in order for the function types to satisfy the subtype relation, and this is exactly what the TCC states.

When a doubly recursive call is found, the inner recursive calls are replaced by variables in the termination TCCs generated for the outer calls. For example, the theory of Figure 3.2 generates the termination TCC of Figure 3.3

where the inner calls to \(f91\) have been replaced by the higher-order variable \(v\), with the recursive signature as shown. Since the obligation forces us to prove the termination condition for all functions whose type is that of \(f91\), it will also hold for \(f91\). This example also illustrates the use of dependent types, discussed in Section 4.6.

In some cases the natural numbers are not a convenient measure; PVS also provides the ordinals, which allow recursion with measures up to \(\varepsilon_0\). This is primarily useful in handling lexicographical orderings. For example, in the definition of the Ackerman function in Figure 3.4,\(^6\) there are two termination TCCs generated (along with a number of subtype TCCs). The first termination TCC is

\[
\text{ack TCC2:}
\]

\[
\text{OBLIGATION}
\]

\[
(\text{FORALL } m, n:\n\]

\[
\text{NOT } m = 0 \text{ AND } n = 0 \text{ IMPLIES ackmeas}(m - 1, 1) < \text{ackmeas}(m, n))
\]

and corresponds to the first recursive call of \(\text{ack}\) in the body of \(\text{ack}\). In this occurrence, it is known that \(m \neq 0\) and \(n = 0\). The remaining expression says that the measure applied to the arguments of the recursive call to \(\text{ack}\) is less than the measure applied to the initial arguments of \(\text{ack}\). Note that the \(<\) in this expression is over the ordinals, not the reals.

---

\(^6\)There are ways of specifying \(\text{ackerman}\) using higher-order functionals, in which case the measure is again on the natural numbers.
parameters are not rewritten; typepred and same-name must be used. These both require
the terms to be given as arguments, making it difficult to automate proofs.

The MACRO declaration is used to indicate definitions that are expanded at typecheck time. Macro
declarations are normal constant declarations, with the MACRO keyword preceding the
type.\footnote{For example, after the declaration
\[ \texttt{N: MACRO nat = 100} \] any reference to \texttt{N} is now automatically replaced by 100, including such forms as \texttt{beta[N]}. Macros are not expanded until they have been typechecked. This is because the name
overloading allowed by PVS precludes expanding during parsing. TCCs are generated before
the definition is expanded.

\section*{3.6 Inductive and Coinductive Definitions}

\textit{Inductive} definitions \cite{1} are used frequently in mathematics. In general, some rules are
given that generate elements of a set, and the inductively defined set is the smallest set
that contains those elements. The obvious example of an inductive definition is the natural
numbers, where the rules are given by Peano’s axioms, with the induction scheme ensuring
that the natural numbers are the smallest set containing 1 and the successor of any natural
number. Language definitions are another example. Most logics have a notion of \textit{formulas},
and these are usually defined inductively.

Paulson \cite{14} notes that this is simply a \textit{least fixedpoint} with respect to a given domain
of elements and a set of rules, which is well-defined if the rules are \textit{monotonic}, by the
well known Knaster-Tarski theorem. From this perspective, the greatest fixedpoint also
exists and corresponds to \textit{coinductive} definitions. Inductive and coinductive definitions are
similar to recursive definitions, in that they have induction principles, and both must satisfy
additional constraints to guarantee that they are well defined.

We will describe inductive definitions first, as they are more familiar. The even integers
provide a simple example of an inductive definition:\footnote{This is similar to the == form of \textit{Ehdm}.}
\begin{verbatim}
  even(n: int): INDUCTIVE bool = n = 0 OR even(n - 2) OR even(n + 2)
\end{verbatim}

With this definition, it is easy to prove, for example, that 0 or 1000 are even, simply by
expanding the definition enough times.\footnote{This is an alternative to the more traditional definition of even? in the prelude.} More is needed, however, in proving general facts,
such as if \( n \) is even, then \( n + 1 \) is not even. To deal with these, we need a means of stating
that an integer is even iff it is so as a result of this definition. In PVS, this is accomplished
by the automatic creation of two induction schemas, that may be viewed using the
M-x prettyprint-expanded command:

\begin{verbatim}
  (apply (repeat (then (expand "even") (flatten) (assert))))
\end{verbatim}

\footnote{In the latter case, \texttt{(apply (repeat (then (expand "even") (flatten) (assert))))} is a good strategy to use, though it should be used with care since it does not terminate on \texttt{even} applied to anything other than an even numeral.}
Declarations

3.6 Inductive and Coinductive Definitions

even_weak_induction: AXIOM
FORALL (P : [int -> boolean]):
  (FORALL (n: int): n = 0 OR P(n - 2) OR P(n + 2) IMPLIES P(n)) IMPLIES
  (FORALL (n: int): even(n) IMPLIES P(n));

even_induction: AXIOM
FORALL (P : [int -> boolean]):
  (FORALL (n: int):
    n = 0 OR even(n - 2) AND P(n - 2) OR even(n + 2) AND P(n + 2)
    IMPLIES P(n))
  IMPLIES (FORALL (n: int): even(n) IMPLIES P(n));

The weak induction axiom states that if $P$ is another predicate that satisfies the even form, then any even number satisfies $P$. Thus even is the smallest such $P$. The second (strong) axiom allows the even predicate to be carried along, which can make proofs easier. These axioms are used by the rule-induct strategy described in the Prover Guide [16].

Inductive definitions are predicates, hence must be functions with eventual range type boolean. For example, in

\[
\begin{align*}
n_1(n,m \text{ int}) & \text{ INDUCTIVE } \text{ int } = n \\
n_2(n,m \text{ int})(x,y \text{ int})(z \text{ int}) & \text{ INDUCTIVE } [\text{ int, int, int } \rightarrow \text{ bool } ] = \\
& \text{ LAMBDA (a,b,c \text{ int}): } n = m \text{ IMPLIES } n_2(n,m)(x,y)(z)(a,b,c)
\end{align*}
\]

$f_1$ is illegal, while $f_2$ returns a boolean value if applied to enough arguments, hence is valid.

To be monotonic, every occurrence of the definition within the defining body must be positive. For this we need to define the parity of an occurrence of a term in an expression $A$: If a term occurs in $A$ with a given parity, then the occurrence retains its parity in $A$ AND $B$, $A$ OR $B$, $B$ IMPLIES $A$, FORALL $y : A$, EXISTS $y : A$, and reverses it in $A$ IMPLIES $B$ and NOT $A$. Any other occurrence is of unknown parity.

The parity of the inductive definition in the definition body is checked, and if some occurrence of the definition is negative, a type error is generated. If some occurrence is of unknown parity, then a monotonicity TCC is generated. For example, given the declarations

\[
\begin{align*}
f : [\text{nat, bool } \rightarrow \text{ bool } ] \\
G(n : \text{nat}) & \text{ INDUCITVE } \text{ bool } = \\
& n = 0 \text{ OR } f(n, G(n-1))
\end{align*}
\]

the monotonicity TCC has the form

\[
\begin{align*}
& (\text{FORALL (P1 : [nat } \rightarrow \text{ boolean ], P2 : [nat } \rightarrow \text{ boolean ]):} \\
& \quad (\text{FORALL (x : nat): P1(x) IMPLIES P2(x)})
\end{align*}
\]

\[
\begin{align*}
\quad \text{IMPLIES} \\
& \quad (\text{FORALL (x : nat):} \\
& \quad \quad x = 0 \text{ OR } f(x, P1(x - 1)) \text{ IMPLIES } x = 0 \text{ OR } f(x, P2(x - 1))));
\end{align*}
\]

Inductive definitions act as constants for the most part, so they may be expanded or used as rewrite rules in proofs. However, they are not usable as auto-rewrite rules, as there is no easy way to determine when to stop rewriting.
To provide induction schemes in the most usable form, they are generated as follows. First, the variables in the definition are partitioned into fixed and non-fixed variables. For example, in the transitive-reflexive closure

\[ TC(R)(x, y) : \text{INDUCTIVE bool} = \]
\[ R(x, y) \lor (\exists z : TC(R)(x, z) \land TC(R)(z, y)) \]

\( R \) is fixed since every occurrence of \( TC \) has \( R \) as an argument in exactly the same position, whereas \( x \) and \( y \) are not fixed. The induction is then over predicates \( P \) that take the non-fixed variables as arguments. If the inductive definition is defined for variable \( V \) partitioned into fixed variables \( F \), and non-fixed variables \( N \), the general form of the (weak) induction scheme is

\[
\text{FORALL } (F, P):
\text{(FORALL } (N):
\text{inductive_body}(N)[P/\text{def}] \text{ IMPLIES } P(N))
\text{ IMPLIES }
\text{(FORALL } (N): \text{def}(V) \text{ IMPLIES } P(N))
\]

In the case of \( TC \), this becomes

\[
\text{TC_weak_induction: AXIOM}
\text{(FORALL } (R: \text{relation, P: }[[T, T] \to \text{boolean]}):
\text{(FORALL } (x: T, y: T):
\text{R}(x, y) \lor (\exists z : (P(x, z) \land P(z, y))) \text{ IMPLIES } P(x, y))
\text{ IMPLIES (FORALL } (x: T, y: T): TC(R)(x, y) \text{ IMPLIES } P(x, y))\);
\]

Coinductive definitions have the same form as inductive definitions, but are introduced with the keyword \text{COINDUCTIVE}, and generate the greatest fix point, rather than the least fix point. The monotonicity conditions are the same, but the coinduction axioms reverse some of the implications. Thus the general form of the (weak) coinduction scheme is

\[
\text{FORALL } (F, P):
\text{(FORALL } (N):
\text{P}(N) \text{ IMPLIES coinductive_body}(N)[P/\text{def}]\)
\text{ IMPLIES }
\text{(FORALL } (N): \text{P}(N) \text{ IMPLIES } \text{def}(V))
\]

As noted earlier, inductive and coinductive definitions are really fixedpoint definitions. For example, the theory in Figure 3.5 shows that an inductive definition is a least fixedpoint, a coinductive definition is a greatest fixpoint, an inductively defined set is a subset of a coindutively defined set, and, if the universe contains a non-wellfounded element, then the coinductively defined set is strictly larger. These results all build on the definitions in the mu-calculus theory of the prelude.

### 3.7 Formula Declarations

Formula declarations introduce \text{axioms, assumptions, theorems,} and \text{obligations}. The identifier associated with the declaration may be referenced in auto-rewrite declarations (see
3.8 Judgements

The facility for defining predicate subtypes is one of the most useful features provided by PVS, but it can lead to a lot of redundant TCCs. Judgements\textsuperscript{11} provide a means for controlling this by allowing properties of operators on subtypes to be made available to the typechecker. There are several kinds of judgements available in PVS. Most of them indicate that an expression belongs to a given type, but the subtype judgement indicates that two types are in the subtype relation.

Number judgement -

Name judgement

Application judgement

Recursive judgement

Expression judgement

Subtype judgement

\textsuperscript{10}The universal closure of a formula is obtained by surrounding the formula with a \texttt{FORALL} binding operator whose bindings are the free variables of the formula. For example, the universal closure of $p(x, y) \Rightarrow q(z)$ is $\forall x, y, z : p(x, y) \Rightarrow q(z)$ (assuming $x$, $y$, and $z$ resolve to variables).

\textsuperscript{11}We prefer this spelling, though many spell checkers do not.
The constant judgement states that a particular constant (or number) has a type more specific than its declared type. The subtype judgement states that one type is a subtype of another.

### 3.8.1 Number and Name Judgements

Number and name judgements

There are two kinds of constant judgements. The simpler kind states that a constant or number belongs to a type different than its declared type.\(^{12}\) For example, the constant judgement declaration

\[
\text{JUDGEMENT } c, \text{ 17 HAS\_TYPE } (\text{prime?})
\]

simply states that the constant \(c\) and the number 17 are both prime numbers. This declaration leads to the TCC formulas \(\text{prime?}(c)\) and \(\text{prime?}(17)\), but in any context in which this declaration is visible, the use of \(c\) or 17 where a prime is expected will not generate TCCs. Thus no TCCs are generated for the formula \(F\) in

\[
\text{RP: } [(\text{prime?}), (\text{prime?}) \rightarrow \text{bool}]
\]

\[
F: \text{FORMULA } \text{RP}(c, 17) \text{ IMPLIES } \text{RP}(17, c)
\]

The second kind of constant judgement is for functions; argument types are provided and the judgement states that when the function is applied to arguments of the given types, then the result has the type following the \text{HAS\_TYPE} keyword. Here is an example that illustrates the need for this kind of judgement:

\[
x, y: \text{VAR } \text{real}
\]

\[
f(x,y): \text{real} = x^x - y^y
\]

\[
n: \text{int} = \text{IF } f(1,2) > 0 \text{ THEN } f(4,3) \text{ ELSE } f(3,2) \text{ ENDIF}
\]

This leads to two TCCs:

\[
n\_\text{TCC1: } \text{OBLIGATION}
\]

\[
f(1, 2) > 0 \text{ IMPLIES }
\]

\[
\text{rational\_pred}(f(4, 3)) \text{ AND integer\_pred}(f(4, 3))
\]

\[
n\_\text{TCC2: } \text{OBLIGATION}
\]

\[
\text{NOT } f(1, 2) > 0 \text{ IMPLIES }
\]

\[
\text{rational\_pred}(f(3, 2)) \text{ AND integer\_pred}(f(3, 2))
\]

The problem here is that although we know that \(f\) is closed under the integers, the typechecker does not. If \(f\) is heavily used, dealing with these TCCs becomes cumbersome. We can try the \textit{ad hoc} solution of adding new overloaded declarations for \(f\):

\[
i, j: \text{VAR } \text{nat}
\]

\[
f(i, j): \text{int} = f(i, j)
\]

But now proofs require an extra definition expansion, and such overloading leads to confusion.\(^{13}\) A more elegant solution is to use a judgement declaration:

\[
f\_\text{int\_is\_int: JUDGEMENT } f(i, j: \text{int}) \text{ HAS\_TYPE } \text{int}
\]

This generates the TCC

\[
f\_\text{int\_is\_int: FORALL } (x: \text{int}, y: \text{int}):
\]

\[
\text{rational\_pred}(f(x, y)) \text{ AND integer\_pred}(f(x, y))
\]

But now the declaration of \(n\) given above generates no TCCs, as the typechecker “knows” that \(f\) is closed on the integers. Note that this is different than the simple judgement

\[
f\_\text{int: JUDGEMENT } f \text{ HAS\_TYPE } [\text{int, int} \rightarrow \text{int}]
\]

In this case, the TCC generated is unprovable:

\(^{12}\)Remember that all numbers are implicitly declared to be of type \text{real}.

\(^{13}\)This is one of the motivations for providing the \texttt{M-x show-expanded-sequent} command.
Declarations

3.8 Judgements

\[ f_{\text{int}}: \text{OBLIGATION} \]
\[ ((\forall x: \text{real}): \text{rational\_pred}(x) \land \text{integer\_pred}(x)) \land \]
\[ (\forall x: \text{real}): \text{rational\_pred}(x) \land \text{integer\_pred}(x)) \]
\[ \land (\forall x1: [\text{real, real}]): \text{rational\_pred}(f(x1)) \land \text{integer\_pred}(f(x1)); \]

A warning is generated when simple constant judgements are declared to be of a function type. In addition, this judgement will not help with the declaration \( n \) above; it can only be used in higher-order functions, for example:

\[ F: [[\text{int, int -> int} \to \text{int}]] \to \text{bool} \]
\[ FF: \text{FORMULA} F(f) \]

The arguments for a function judgement follow the syntax for function declarations; so a curried function may be given multiple judgements:

\[ f(x, y: \text{real})(z: \text{real}) : \text{real} \]
\[ f_{\text{closed}}: \text{JUDGEMENT} f(x, y: \text{nat})(z: \text{int}) \text{ HAS\_TYPE int} \]
\[ f_{\text{2\_closed}}: \text{JUDGEMENT} f(x, y: \text{int}) \text{ HAS\_TYPE [real -> int]} \]

If a constant judgement declaration specifies a name, it must refer to a unique constant and its type must be compatible with the type expression following the HAS\_TYPE keyword. If it is a number, then its type must be compatible with the number type.

Constant judgements generally lead to TCCs. If no TCC is generated, then the judgement is not actually needed, and a warning to this effect is produced. Simple (non-functional) constant judgements generate TCCs indicating that the constant belongs to the specified type. Constant function judgements generate TCCs that reflect closure conditions.

The judgement facility cannot be used to remove all redundant TCCs; the variables used for arguments must be unique, and full expressions may not be included. Hence the following are not legal:

\[ x: \text{VAR real} \]
\[ x\_\times\_x\_is\_nonneg: \text{JUDGEMENT} *(x, x) \text{ HAS\_TYPE nonneg\_real} \]
\[ c: \text{real} \]
\[ x\_\times\_c\_is\_even: \text{JUDGEMENT} *(x, c) \text{ HAS\_TYPE (even?)} \]

3.8.2 Subtype Judgements

The subtype judgement is used to fill in edges of the subtype graph that otherwise are unknown to the typechecker. For example, consider the following declarations:

\[ \text{nonzeroreal: NONEMPTY\_TYPE} = \{r: \text{real} | r \neq 0\} \text{ CONTAINING 1} \]
\[ \text{rational: NONEMPTY\_TYPE FROM real} \]
\[ \text{nneg\_rat: NONEMPTY\_TYPE} = \{r: \text{rational} | r \geq 0\} \text{ CONTAINING 0} \]
\[ \text{posrat: NONEMPTY\_TYPE} = \{r: \text{nneg\_rat} | r > 0\} \text{ CONTAINING 1} \]
\[ /: [\text{real, nonzero\_real -> real}] \]

For \( r \) of type real and \( q \) of type posrat, the expression \( r/q \) leads to the TCC \( q \neq 0 \). One solution, if \( q \) is a constant, is to use a constant judgement as described above. But if there are many constants involving the type posrat, this requires a lot of judgement declarations, and does not help at all for variables or compound expressions. The subtype judgement solves this by stating that posrat is a subtype of nzrat. Another subtype judgement states that nzrat is a subtype of nzreal:

14Earlier versions of PVS simply interpreted this form as a closure condition, but this is less flexible.
With these judgements, TCCs will not be generated for any denominator that is of type `posrat`. With the (prelude) judgement declarations

\[
\begin{align*}
\text{JUDGEMENT posrat SUBTYPE_OF nzrat} \\
\text{JUDGEMENT nzrat SUBTYPE_OF nzreal}
\end{align*}
\]

not only are there no TCCs generated for \(r/q\), but none are generated for \(r/(q + 2)\), \(r/((q + 2) \times q)\), etc.

Given a subtype judgement declaration of the form

\[
\text{JUDGEMENT S SUBTYPE_OF T}
\]

it is an error if \(S\) is already known to be a subtype of \(T\), or if they are not compatible. Otherwise, \(T\) must be of the form \(\{x: ST \mid p(x)\}\), where \(ST\) is the least compatible type of \(S\) and \(T\), and a TCC will be generated of the form \(\text{FORALL } (x:S): p(x)\). Remember that subtyping on functions only works on range types, so the subtype judgement

\[
\text{JUDGEMENT [nat -> nat] SUBTYPE_OF [int -> int]}
\]

leads to the unprovable TCC

\[
\text{FORALL } (x1:nat, y1:int): y1 >= 0 \text{ AND TRUE}
\]

### 3.8.3 Judgement Processing

When a judgement declaration is typechecked, TCCs are generated as explained above and the judgement is added to the current context for use in typechecking expressions. The typechecker typechecks expressions in two passes; in the first pass it simply collects possible types for subexpressions, and in the second pass it recursively tries to determine a unique type based on the expected type, and generates TCCs accordingly; this is where judgements are used. If the expression is a constant (name or number), then all non-functional judgements are collected for that constant and used to generate a minimal TCC relative to the expected type.

If it is an application whose operator is a name, then functional judgements of the corresponding arity are collected for the operator, and those judgements for which the application arguments are all known to be of the corresponding judgement argument types are extracted, and a minimal TCC is generated from these.

In addition to inhibiting the generation of TCCs during typechecking, judgements are also important to the prover; they are used explicitly in the `typepred` command, and implicitly in `assert`, where the judgement type information is provided to the ground decision procedures.

Subtype judgements are used in determining when one type is a subtype of another, which is tested frequently during typechecking and proving, including in the test on argument types described above.

### 3.9 Conversions

Conversions are functions that the typechecker can insert automatically whenever there is a type mismatch. They are similar to the implicit coercions for converting integers to floating
point used in many programming languages. PVS provides some builtin conversions in the prelude, but conversions may also be provided by the user using conversion declarations. A conversion declaration consists of the keyword CONVERSION, optionally followed by ‘+’ or ‘-’ and an expression. CONVERSION+ is equivalent to CONVERSION. The expression must be of type a (subtype of) a function type, where the domain and range are not compatible. This is because conversions are only triggered when there would otherwise be a type error, and compatible types may lead to unproveable TCCs, but not to type errors. Judgements are the proper way to control the generation of TCCs, see Section 3.8 for details.

3.9.1 Conversion Examples

Here is a simple example.

c: [int -> bool]
CONVERSION c
two: FORMULA 2

Here, since formulas must be of type boolean, the typechecker automatically invokes the conversion and changes the formula to c(2). This is done internally, and is only visible to the user on explicit command\(^\text{15}\) and in the proof checker.

A more complex conversion is illustrated in the following example.

g: [int -> int]
F: [[nat -> int] -> bool]
F_app: FORMULA F(g)

As this stands, F_app is not type-correct, because a function of type [int -> int] is supplied where one of type [nat -> int] is required, and PVS requires equality on domain types for function types to be compatible. However it is clear that g naturally induces a function from nat to int by simply restricting its domain. Such a domain restriction is achieved by the restrict conversion that is defined in the PVS prelude as follows:

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

The construction $S: \text{TYPE FROM } T$ specifies that the actual parameter supplied for $S$ must be a subtype of the one supplied for $T$. The specification states that restrict$(f)$ is a function from $S$ to $R$ whose values agree with $f$ (which is defined on the larger domain $T$). Using this approach, a type correct version of $F_app$ can be written as $F(\text{restrict}[\text{int}, \text{nat}, \text{int}](g))$. This provides the convenience of contravariant subtyping, but without the inherent complexity (in particular, with contravariant subtyping the type of equality must be correct in substituting equals for equals, making proofs less perspicuous).

\(^\text{15}\)The M-x prettyprint-expanded command.
It is not so obvious how to expand the domain of a function in the general case, so this approach does not work automatically in the other direction. It does, however, work well for the important special case of sets (or, equivalently, predicates): a set on some type $S$ can be extended naturally to one on a supertype $T$ by assuming that the members of the type-extended set are just those of the original set. Thus, if $\text{extend}(s)$ is the type-extended version of the original set $s$, we have $\text{extend}(s)(x) = s(x)$ if $x$ is in the subtype $S$, and $\text{extend}(s)(x) = \text{false}$ otherwise. We can say that false is the “default” value for the type-extended function. Building on this idea, we arrive at the following specification for a general type-extension function.

\[
\begin{align*}
\text{extend} \ [T: \text{TYPE}, S: \text{TYPE FROM T}, R: \text{TYPE}, d: R]: \text{THEORY} \\
\text{BEGIN} \\
\quad f: \text{VAR} \ [S \rightarrow R] \\
\quad t: \text{VAR} \ T \\
\quad \text{extend}(f)(t): R = \text{IF} \ S_{\text{pred}}(t) \ \text{THEN} \ f(t) \ \text{ELSE} \ d \ \text{ENDIF} \\
\text{END} \ extend
\end{align*}
\]

The function $\text{extend}(f)$ has type $[T \rightarrow R]$ and is constructed from the function $f$ of type $[S \rightarrow R]$ (where $S$ is a subtype of $T$) by supplying the default value $d$ whenever its argument is not in $S$ ($S_{\text{pred}}$ is the recognizer predicate for $S$). Because of the need to supply the default $d$, this construction cannot be applied automatically as a conversion. However, as noted above, false is a natural default for functions with range type bool (i.e., sets and predicates), and the following theory establishes the corresponding conversion.

\[
\begin{align*}
\text{extend}_\text{bool} \ [T: \text{TYPE}, S: \text{TYPE FROM T}]: \text{THEORY} \\
\text{BEGIN} \\
\quad \text{CONVERSION} \ \text{extend}[T, S, \text{bool}, \text{false}] \\
\text{END} \ extend_\text{bool}
\end{align*}
\]

In the presence of this conversion, the type-incorrect formula $B_{\text{app}}$ in the following specification

\[
\begin{align*}
\quad b: [\text{nat} \rightarrow \text{bool}] \\
\quad B: [[\text{int} \rightarrow \text{bool}] \rightarrow \text{bool}] \\
\quad B_{\text{app}}: \text{FORMULA} \ B(b)
\end{align*}
\]

is automatically transformed to $B(\text{extend}[\text{int}, \text{nat}, \text{bool}, \text{false}](b))$.

### 3.9.2 Lambda conversions

Conversions are also useful (for example, in semantic encodings of dynamic or temporal logics) in “lifting” operations to apply pointwise to sequences over their argument types. Here is an example, where $\text{state}$ is an uninterpreted (nonempty) type, and a state variable $v$ of type real is represented as a constant of type $[\text{state} \rightarrow \text{real}]$. 

30
Declarations

3.9 Conversions

\[
\text{th: THEORY}
\begin{align*}
\text{BEGIN} & \\
\text{CONVERSION+ K\_conversion} & \\
\text{state: TYPE+} & \\
l: [\text{state -> list[int]}] & \\
x: [\text{state -> real]} & \\
b: [\text{state -> bool}] & \\
bv: \text{VAR [state -> bool]} & \\
s: \text{VAR state} & \\
\text{box(bv): bool = FORALL s: bv(s)} & \\
\text{F1: FORMULA box(x > 1)} & \\
\text{F2: FORMULA box(b IMPLIES length(l) + 3 > x)} & \\
\text{END th}
\end{align*}
\]

In this example, the formulas \( F1 \) and \( F2 \) are not type correct as they stand, but with a \textit{lambda conversion}, triggered by the K\_conversion in the PVS prelude, these formulas are converted to the forms

\[
\begin{align*}
\text{F1: FORMULA box(LAMBDA (x1: state): x(x1) > 1)} & \\
\text{F2: FORMULA box(LAMBDA (x3: state): b(x3) IMPLIES (LAMBDA (x2: state): (LAMBDA (x1: state): length(l(x)))(x1) + 3)(x2) > x(x2))(x3))}
\end{align*}
\]

3.9.3 Conversions on Type Constructors

Conversions for record, tuple, and function types may be found componentwise, without having to create the corresponding conversion declaration. Here is an example.

\[
\begin{align*}
\text{bi: [bool -> int]} & \\
\text{ib: [int -> bool]} & \\
\text{CONVERSION+ bi, ib} & \\
t: [\text{int, int, int}] = (\text{true, false, 3}) & \\
r: [# a, b: int #] = (# a := true, b := false #) & \\
f: [\text{int, int -> int}] = \text{AND}
\end{align*}
\]

With conversions displayed, this becomes the following.
3.9 Conversions

This section discusses how conversions are declared and processed in the context of type checking.

### 3.9.4 Conversion Processing

In general, conversions are applied by the typechecker whenever it would otherwise emit a type error. In the simplest case, if an expression \( e \) of type \( T_1 \) occurs where an incompatible type \( T_2 \) is expected, the most recent compatible conversion \( C \) is found in the context and the occurrence of \( e \) is replaced by \( C(e) \).

Conversions are ordered in the context; if multiple compatible conversions are available, the most recently declared conversion is used. Hence, in

```
CONVERSION c1

IMPORTING th1, th2

CONVERSION c2

F: FORMULA 2
```

For formula \( F \), \( c_2 \) is the most recent conversion, followed by the conversions in theory \( th_2 \), those in \( th_1 \), and finally \( c_1 \). Note that the relative order of the constant declarations (e.g., \( c_1 \) and \( c_2 \) above) doesn’t matter, only the \textsc{conversion} declarations.

When conversions are available on either the argument(s) or the operator of an application, the arguments get precedence.

For an application \( e(x_1, \ldots, x_n) \) the possible types of the operator \( e \), and the arguments \( x_i \) are determined, and for each operator type \([D_1, \ldots, D_n \to R]\) and argument type \( T_i \), if \( D_i \) is not compatible with \( T_i \), conversions of type \([T_i \to D_i]\) are collected. If such conversions are found for every argument that doesn’t have a compatible type, then those conversions are applied. Otherwise an operator conversion is looked for.

Note that compositions of conversion are never searched for, as this would slow down processing too much. If you want to use a composition, include a conversion declaration for it. Here is an example:
In this example, F1 leads to a type error, but when we make the composition a conversion, the same expression in F2 applies the conversion rather than give a type error.

### 3.9.5 Conversion Control

As stated above, conversions are only applied when typechecking otherwise fails. In some cases, a conversion can allow a specification to typecheck, but the meaning is different than what was intended. This is most likely for the K_conversion, which was introduced when the mu calculus theory was added to the prelude in support of the model checker. When a conversion is applied that fact is noted as a message, and may be viewed using the show-theory-messages command. However, these messages are easily overlooked, so instead PVS allows finer control over conversions.

Thus in addition to the CONVERSION form, the CONVERSION- form is available allowing conversions to be turned off. For uniformity, the CONVERSION+ form is also available as an alias for CONVERSION. CONVERSION- disables conversions.

The following theory illustrates the idea:

```
t1: THEORY
BEGIN
  c: [int -> bool]
  CONVERSION+ c
  f1: FORMULA 3
  CONVERSION- c
  f2: FORMULA 3
END t1
```

Here f2 leads to a type error.

Another example is provided by the definition of the CTL temporal operators in the prelude theory ctops, which are surrounded by CONVERSION+ and CONVERSION- declarations that first enable the K_conversion then disable it at the end of the theory. All other conversions declared in the prelude remain enabled. They may be disabled within any theory by using the CONVERSION- form.

When theories containing conversion declarations are imported, the conversions are imported as well. Thus if t2 enables the c declaration without subsequently disabling it, then IMPORTING t1, t2 would enable the conversion, but IMPORTING t2, t1 would leave it disabled.
Conversion declarations may be generic or instantiated. This allows, for example, enabling the generic form of a conversion while disabling particular instances.

### 3.10 Library Declarations

Library declarations are used to introduce a new PVS context into a specification. Thus a specification may be developed in one context, and used in many other contexts. This provides more flexibility, at the cost of less portability. Any PVS context other than the current one may be considered a library.

An example of a library declaration is:

```
lib: LIBRARY = "~/pvs/protocols"
```

When encountered, the system verifies that the directory specified within the quotation marks exists, and that it has a PVS context file (.pvscontext). The library declaration is made use of by including the library id in an importing name:

```
IMPORTING lib@sliding_window[n]
```

This has the effect of bringing in the `sliding_window` theory, exactly as if the theory belonged to the current context.

There are several libraries distributed with PVS, in the directory `lib`. It is not necessary to give a library declaration for libraries in this directory, as it will be automatically searched for library importings. Also, as described in the PVS System Guide, any libraries found on the environment variable `PVS_LIBRARY_PATH` do not need library declarations. For example, to import the finite sets library over the natural numbers:

```
IMPORTING finite_sets@finite_sets[nat]
```

An alternative approach (described in the *PVS User Guide*) is to use the `M-x load-prelude-library`, which augments the PVS prelude with the theories from a given context.

### 3.11 Auto-rewrite Declarations

One of the problems with writing useful theories or libraries is that there is no easy way to convey how the theory is to be used, other than in comments or documentation. In particular, the specifier of a theory usually knows which lemmas should always be used as rewrites, and which should never appear as rewrites. Auto-rewrite declarations allow for both forms of control. Those that should always be used as rewrites are declared with the `AUTO_REWRITE+` or `AUTO_REWRITE` keyword, and those that should not are declared with `AUTO_REWRITE-`. These will be referred to as *auto-rewrites* and *stop-auto-rewrites* below.

When a proof is initiated for a given formula, all of the auto-rewrite names in the current context that haven’t subsequently been removed by stop-auto-rewrite declarations are collected and added to the initial proof state. The stop-auto-rewrite declaration, in addition to removing auto-rewrite names, also affects the following commands described in the Prover manual:

- `auto-rewrite-theory`,
- `auto-rewrite-theories`,
- `auto-rewrite-theory-with-importings`,
- `auto-rewrite-lemmas`,
• simplify-with-rewrites,
• autorewrite-defs,
• install-rewrites,
• auto-rewrite-explicit,
• grind,
• inductand-simplify,
• measure-induct-and-simplify, and
• model-check

These commands collect all definitions and formulas except those that appear in AUTO_REWRITE- declarations. Thus suppose a theory \( T \) contains the lemmas \( \text{lem1} \), \( \text{lem2} \), and \( \text{lem3} \) and the declarations

\[
\text{AUTO_REWRITE+ lem1} \\
\text{AUTO_REWRITE- lem3}
\]

Then in proving a formula of a theory that imports \( T \), \( \text{lem1} \) is initially an auto-rewrite, and the command (\text{auto-rewrite-theory "T"}) will additionally install \( \text{lem2} \). To auto-rewrite with \( \text{lem3} \), simply use (\text{auto-rewrite "lem3"}). To exclude \( \text{lem1} \), use (\text{stop-auto-rewrite "lem1"}) or (\text{auto-rewrite-theory "T" :exclude "lem1"}).

The autorewrites theory shows a simple example.

```plaintext
autorewrites: THEORY
BEGIN
  AUTO_REWRITE+ zero_times3
  a, b: real
  f1: FORMULA a * b = 0 AND a /= 0 IMPLIES b = 0
  AUTO_REWRITE- zero_times3
  f2: FORMULA a * b = 0 AND a /= 0 IMPLIES b = 0
END autorewrites
```

Here \( f1 \) may be proved using only assert, but \( f2 \) requires more.

Rewrite names may have suffixes, for example, \text{foo}! or \text{foo}!!. Without the suffix, the rewrite is \textit{lazy}, meaning that the rewrite will only take place if conditions and TCCs simplify to true. A condition in this case is a top-level IF or CASES expression. With a single exclamation point the auto-rewrite is \textit{eager}, in which case the conditions are irrelevant, though if it is a function definition it must have all arguments supplied. With two exclamation points it is a \textit{macro} rewrite, and terms will be rewritten even if not all arguments are provided. See the prover guide for more details; the notation is derived from the prover commands \text{auto-rewrite}, \text{auto-rewrite!}, and \text{auto-rewrite!!}.

In addition, a rewrite name may be disambiguated by stating that it is a formula, or giving its type if it is a constant. Without this any definition or lemma in the context with the same name will be installed as an auto-rewrite.

In order to be more uniform, these new forms of name are also available for the auto-rewrite prover commands. Thus the command
(auto-rewrite "A" ("B" "-2") "C" (("1" "D")))

may now be given instead as

(auto-rewrite "A" "B!" "-2!" "C" "1!!" "D!!")

The older form is still allowed, but is deprecated, and may not be mixed with the new form. Notice that in the auto-rewrite commands formula numbers may also be used, and these may be followed by exclamation points, but not by a formula keyword or type.
### Declarations

#### 3.11 Auto-rewrite Declarations

<table>
<thead>
<tr>
<th>Rule</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LibDecl</strong></td>
<td>:∶= Ids : LIBRARY [ = ] String</td>
</tr>
<tr>
<td><strong>TheoryDecl</strong></td>
<td>:∶= Ids[ DeclFormals ] : THEORY = TheoryDeclName</td>
</tr>
<tr>
<td><strong>TypeDecl</strong></td>
<td>:∶= Id[ DeclFormals ] [</td>
</tr>
<tr>
<td><strong>VarDecl</strong></td>
<td>:∶= IdOps[ DeclFormals ] : VAR TypeExpr</td>
</tr>
<tr>
<td><strong>ConstDecl</strong></td>
<td>:∶= IdOp[ DeclFormals ] [</td>
</tr>
<tr>
<td><strong>RecursiveDecl</strong></td>
<td>:∶= IdOp[ DeclFormals ] [</td>
</tr>
<tr>
<td><strong>MacroDecl</strong></td>
<td>:∶= IdOp[ DeclFormals ] [</td>
</tr>
<tr>
<td><strong>InductiveDecl</strong></td>
<td>:∶= IdOp[ DeclFormals ] [</td>
</tr>
<tr>
<td><strong>CoInductiveDecl</strong></td>
<td>:∶= IdOp[ DeclFormals ] [</td>
</tr>
<tr>
<td><strong>FormulaDecl</strong></td>
<td>:∶= Ids[ DeclFormals ] : FormulaName Expr</td>
</tr>
<tr>
<td><strong>Judgement</strong></td>
<td>:∶= SubtypeJudgement</td>
</tr>
<tr>
<td><strong>SubtypeJudgement</strong></td>
<td>:∶= [ IdOp[ DeclFormals ] : ] JUDGEMENT TypeExpr † , SUBTYPE_OF TypeExpr</td>
</tr>
<tr>
<td><strong>ConstantJudgement</strong></td>
<td>:∶= [ IdOp[ DeclFormals ] : ] [ RECURSIVE ] JUDGEMENT ConstantReference † , HAS_TYPE TypeExpr</td>
</tr>
<tr>
<td><strong>ConstantReference</strong></td>
<td>:∶= Name Bindings *</td>
</tr>
<tr>
<td><strong>Conversion</strong></td>
<td>:∴= { CONVERSION</td>
</tr>
<tr>
<td><strong>RewriteName</strong></td>
<td>:∴= Name [ ! [ ! ] ] [ : { TypeExpr</td>
</tr>
<tr>
<td><strong>DeclFormals</strong></td>
<td>:∴= [ DeclFormal † , ]</td>
</tr>
<tr>
<td><strong>DeclFormal</strong></td>
<td>:∴= TheoryFormalType</td>
</tr>
<tr>
<td><strong>Bindings</strong></td>
<td>:∴= ( Binding † , )</td>
</tr>
<tr>
<td><strong>Binding</strong></td>
<td>:∴= TypedId</td>
</tr>
<tr>
<td><strong>TypedIds</strong></td>
<td>:∴= IdOps [ : TypeExpr ] [</td>
</tr>
<tr>
<td><strong>TypedId</strong></td>
<td>:∴= IdOp [ : TypeExpr ] [</td>
</tr>
</tbody>
</table>

**Figure 3.1: Declarations Syntax**
3.11 Auto-rewrite Declarations

\[ \text{f91: THEORY} \]
\[ \text{BEGIN} \]
\[ i: \text{VAR nat} \]
\[ \text{f91}(i): \]
\[ \text{RECURSIVE \{j: \text{nat} | \text{IF } i > 100 \text{ THEN } j = i - 10 \text{ ELSE } j = 91 \text{ ENDIF} \} =} \]
\[ \text{(IF } i > 100 \text{ THEN } i - 10 \text{ ELSE } \text{f91}(\text{f91}(i + 11)) \text{ ENDIF)} \]
\[ \text{MEASURE (LAMBDA } i: \text{IF } i > 101 \text{ THEN } 0 \text{ ELSE } 101 - i \text{ ENDIF)} \]
\[ \text{END f91} \]

Figure 3.2: Theory f91

\[ \text{f91 TCC5: OBLIGATION} \]
\[ \text{FORALL (i: \text{nat},} \]
\[ \text{v: [i1:} \]
\[ \{z: \text{nat} |} \]
\[ \text{(IF } z > 101 \text{ THEN } 0 \text{ ELSE } 101 - z \text{ ENDIF}) <} \]
\[ \text{(IF } i > 101 \text{ THEN } 0 \text{ ELSE } 101 - i \text{ ENDIF}) \} \rightarrow} \]
\[ \{j: \text{nat} | \text{IF } i1 > 100 \text{ THEN } j = i1 - 10 \text{ ELSE } j = 91 \text{ ENDIF}\}]:} \]
\[ \text{NOT } i > 100 \text{ IMPLIES} \]
\[ \text{IF } i > 100 \text{ THEN } v(v(i + 11)) = i - 10 \text{ ELSE } v(v(i + 11)) = 91 \text{ ENDIF}; \]

Figure 3.3: Termination TCC for f91

\[ \text{ackerman: THEORY} \]
\[ \text{BEGIN} \]
\[ m, n: \text{VAR nat} \]
\[ \text{ackmeas}(m, n): \text{ordinal} = \]
\[ \text{(IF } m = 0 \text{ THEN zero} \]
\[ \text{ELSIF } n = 0 \text{ THEN add}(m, \text{add}(1, \text{zero}, \text{zero}), \text{zero}) \]
\[ \text{ELSE add}(m, \text{add}(1, \text{zero}, \text{zero}), \text{add}(n, \text{zero}, \text{zero})) \]
\[ \text{ENDIF}) \]
\[ \text{ack}(m, n): \text{RECURSIVE nat} = \]
\[ \text{(IF } m = 0 \text{ THEN } n + 1 \]
\[ \text{ELSIF } n = 0 \text{ THEN } \text{ack}(m - 1, 1) \]
\[ \text{ELSE add}(m - 1, \text{ack}(m, n - 1)) \]
\[ \text{ENDIF}) \]
\[ \text{MEASURE ackmeas} \]
\[ \text{END ackerman} \]

Figure 3.4: Theory ackerman
begin
N: TYPE+
n, m: VAR N
0: N
S: [N -> N]
Sax1: AXIOM 0 /= S(n)
Sax2: AXIOM S(m) = S(n) => m = n
% Assume a non-wellfounded element
nwf_exists: AXIOM EXISTS n: n = S(n)
Nind(n): INDUCTIVE bool = n = 0 OR EXISTS m: n = S(m) & Nind(m)
Ncoind(n): COINDUCTIVE bool = n = 0 OR EXISTS m: n = S(m) & Ncoind(m)
% NN is the predicate transformer corresponding to the (co)inductive defs
NN(p: pred[N])(n): bool = n = 0 OR EXISTS m: n = S(m) & p(m)
% These use the lfp and gfp defs from the prelude mucalculus theory
ind_lfp: FORMULA Nind = lfp(NN)
coid_gfp: FORMULA Ncoind = gfp(NN)
% Repeat Nind_weak_induction, which is proved from lfp_induction
Nind_weak_induction_repeated: FORMULA
  FORALL (P: [N -> boolean]): (FORALL (n): (n = 0 OR (EXISTS m: n = S(m) & P(m))) IMPLIES P(n))
  IMPLIES (FORALL (n): Nind(n) IMPLIES P(n));
% Inductive definitions are a subset of coinductive
ind_sub_co: FORMULA Nind(n) => Ncoind(n)
% Because there is a non-wellfounded element, we can show that
% the coinductive set is larger.
co_has_more: FORMULA EXISTS n: Ncoind(n) & NOT Nind(n)
end inductive_fixpoint

Figure 3.5: Inductive definitions and fixpoints
Chapter 4

Types

PVS specifications are strongly typed, meaning that every expression has an associated type (although it need not be unique, more on this later). The PVS type system is based on structural equivalence instead of name equivalence, so types are closely related to sets, in that two types are equal iff they have the same elements. Section 3.1 describes the introduction of type names, which are the simplest type expressions. More complex type expressions are built from these using type constructors. There are type constructors for subtypes, function types, tuple types, cotuple types, and record types. Function, record, and tuple types may also be dependent. A form of type application is provided that makes it more convenient to specify parameterized subtypes. There are also provisions for creating abstract datatypes, described in Chapter 9.

Type expressions occur throughout a specification; in particular, they may appear in theory parameters, type declarations, variable declarations, constant declarations, recursive and inductive definitions, conversions, and judgements. In addition, they may appear in certain expressions (coercions and local bindings, see pages 60 and 54, respectively), and as actual parameters in names (page 77). In the many examples which follow, type expressions will be presented in the context of type declarations; but it must be remembered that they can appear in any of the above places.

4.1 Subtypes

Any collection of elements of a given type itself forms a type, called a subtype. The type from which the elements are taken is called the supertype. The elements which form the subtype are determined by a subtype predicate on the supertype.

Subtypes in PVS provide much of the expressive power of the language, at the cost of making typechecking undecidable. There are two forms of subtypes. The first is similar to the notation used to define a set:

\[ t: \text{TYPE} = \{x: s \mid p(x)\} \]

where \( p \) is a predicate on the type \( s \).\(^1\) This has the usual set-theoretical meaning, since

\(^1\)If \( x \) has been previously declared as a variable of type \( s \), then the “: \( s \)” may be omitted.
4.1 Subtypes

Types in PVS are modeled as sets. Subtypes may also be presented in an abbreviated form, by giving a predicate surrounded by parentheses:

\[ \tau : \text{TYPE} = (p) \]

This is equivalent to the form above.

Note that if the predicate \( p \) is everywhere false, then the type is empty. PVS supports empty types, and the term type is used to refer to any type, including the empty type. This is discussed in Section 3.1 (page 12).

Subtypes tend to make specifications more succinct and easier to read. For example, in a specification such as

\[ \text{FORALL} (i:\text{int}): (i >= 0 \ IMPLIES (\text{EXISTS} (j:\text{int}): j >= 0 \ AND \ j > i)) \]

it is much more difficult to see what is being stated than in the equivalent

\[ \text{FORALL} (i:\text{nat}): (\text{EXISTS} (j:\text{nat}): j > i)) \]

where \( \text{nat} \) is defined in the prelude as

\[ \text{naturalnumber}: \text{NONEMPTY\_TYPE} = \{i:\text{integer} | i >= 0\} \text{ CONTAINING 0} \]

\[ \text{nat}: \text{NONEMPTY\_TYPE} = \text{naturalnumber} \]

Subtype constructors consist of a supertype and a subtype predicate on the supertype. The primary property of a subtype is that any element which belongs to the subtype auto-
matically belongs to the supertype. In addition, functions defined on a type automatically apply to the subtype.

There are two type-correctness conditions (TCCs) associated with subtypes. The first concerns empty types as described in section 3.1.7. The second TCC associated with subtypes is the subtype TCC, which comes about from the use of operations defined on subtypes that are applied to elements of the supertype. By this means partial functions may be handled directly, without recourse to a partial term logic or some form of multi-valued logic. For instance, division in PVS is a total function, with signature \([\text{real}, \text{nonzero\_real} \rightarrow \text{real}]\). So given the formula

\[
\text{div\_form: \text{FORMULA}} \quad \text{FORALL} \quad (x, y: \text{int}): \quad x \neq y \implies (x - y)/(y - x) = -1
\]

the denominator is of type integer, but the signature for \(\text{/}\) demands a nonzero\_real. The typechecker thus generates a subtype TCC whose conclusion is \((y - x) \neq 0\). The premises of the TCC are obtained from the expressions context—the conditions which lead to the / operator—in this case \(x \neq y\).\(^2\) The TCC is then

\[
\text{div\_form\_TCC1: \text{OBLIGATION}}
\]

\[
(\text{FORALL} \quad (x,y: \text{int}): \quad x \neq y \implies (y - x) \neq 0)
\]

which is easily discharged by the prover. In general, the context of an expression is obtained from expressions involving IF-THEN-ELSE, AND, OR, and IMPLIES by translating to the IF-THEN-ELSE form. Specifically,

<table>
<thead>
<tr>
<th>Expression</th>
<th>Context for e</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF a THEN e ELSE c ENDIF</td>
<td>a</td>
</tr>
<tr>
<td>IF a THEN b ELSE e ENDIF</td>
<td>NOT a</td>
</tr>
<tr>
<td>a AND e</td>
<td>a</td>
</tr>
<tr>
<td>a OR e</td>
<td>NOT a</td>
</tr>
<tr>
<td>a IMPLIES e</td>
<td>a</td>
</tr>
</tbody>
</table>

Note that only these operators are treated this way; if, for example, IMPLIES is overloaded it will not include the left-hand side in the context for typechecking the right-hand side. The TCCs generated from the context of expression involving a subtype are sufficient, but not necessary conditions that ensure that the value of the expression does not depend on the value of functions applied outside their domain.

Subtype TCCs may occur anywhere there is a mismatch between the type of a term and the use of it, not just in function applications. For example, the following use of record types leads to an unprovable subtype TCC.

\[
r: [\# a, b: \text{nint} #] = (\# a := 0, b := 0 #)
\]

### 4.2 Function Types

Function types have three equivalent forms:

- \([t_1, \ldots, t_n \rightarrow t]\)

\(^2\)As described in the Formal Semantics [12], the context containing declarations is extended to allow boolean expressions.
4.3 Tuple Types

- **FUNCTION**[t₁, ..., tₙ --> t]
- **ARRAY**[t₁, ..., tₙ --> t]

where each tᵢ is a type expression. An element of this type is simply a function whose domain is the sequence of types t₁, ..., tₙ, and whose range is t. A function type is empty if the range is empty and the domain is not. There is no difference in meaning between these three forms; they are provided to support different intensional uses of the type, and may suggest how to handle the given type when an implementation is created for the specification.

The two forms pred[t] and setof[t] are both provided in the prelude as shorthand for [t --> bool]. There is no difference in semantics, as sets in PVS are represented as predicates. The different keywords are provided to support different intentions; pred focuses on properties while setof tends to emphasize elements.

A function type [t₁, ..., tₙ --> t] is a subtype of [s₁, ..., sₙ --> s] iff s is a subtype of t, n = m, and sᵢ = tᵢ for 1 ≤ i ≤ n. This leads to subtype TCCs (called domain mismatch TCCs) that state the equivalence of the domain types. For example, given

p, q: pred[int]
f: [{x: int | p(x)} --> int]
g: [{x: int | q(x)} --> int]
h: [int --> int]
eq1: FORMULA f = g
eq2: FORMULA f = h

The following TCCs are generated:
eq1_TCC1: OBLIGATION
(FORALL (x1: {x : int | q(x)}, y1 : {x : int | p(x)) : q(y1) AND p(x1))
eq2_TCC1: OBLIGATION
(FORALL (x1: int, y1 : {x : int | p(x)) : TRUE AND p(x1))

Section 3.9.1 on page 29 explains how the restrict conversion may be automatically applied in some cases to eliminate the production of these TCCs.

4.3 Tuple Types

Tuple types (also called product types) have the form [t₁, ..., tₙ], where the tᵢ are type expressions. Note that the 0-ary tuple type is not allowed. Elements of this type are tuples whose components are elements of the corresponding type. For example, (1, TRUE, (LAMBDA (x:int): x + 1)) is an expression of type [int, bool, [int --> int]]. Order is important. Associated with every n-tuple type is a set of projection functions: `₁, `₂, ..., (or proj_1, proj_2, ...) where the ith projection is of type [[t₁, ..., tₙ] --> tᵢ]. A tuple type is empty if any of its component types is empty. Function type domains and tuple types are closely related, as the types [t₁, ..., tₙ --> t] and [[t₁, ..., tₙ] --> t] are equivalent; see Section 5.9 for more details.
4.4 Cotuple Types

Cotuple types (also called coproduct or sum types) provide a way to form the disjoint union of types. The syntax is similar to that for tuple types, but with ‘+’ in place of ‘,’ so have the form \([t_1 + \ldots + t_n]\). Elements of this type are essentially pairs consisting of an index and a value for the type corresponding to the index. In PVS the syntax for this is \(\text{IN}_i(e)\), where \(e\) is an expression of type \(t_i\). For example, \(\text{IN}_2(3)\) is an expression of type \([\text{bool} + \text{int} + \text{int -> int}]\), or any other cotuple type whose second component type contains 3. A cotuple type is empty iff all its component types are empty.

cT: TYPE = [int + bool + [int -> int]]

Associated with a cotuple type are injections \(\text{IN}_i\), predicates \(\text{IN?}_i\), and extractions \(\text{OUT}_i\) (none of these is case-sensitive). For example, in this case we have

\[
\begin{align*}
\text{IN}_1: & \ [\text{int -> cT}] \\
\text{IN?}_1: & \ [\text{cT -> bool}] \\
\text{OUT}_1: & \ [(\text{IN?}_1) -> \text{int}]
\end{align*}
\]

Thus \(\text{IN}_2(\text{true})\) creates a \(\text{cT}\) element, and an arbitrary \(\text{cT}\) element \(c\) is processed using \text{CASES}, e.g.,

\[
\text{CASES } c \ \text{OF} \\
\text{IN}_1(i): i + 1, \\
\text{IN}_2(b): \text{IF } b \text{ THEN } 1 \text{ ELSE } 0 \text{ ENDIF,} \\
\text{IN}_3(f): f(0)
\]

\text{ENDCASES}

This is very similar to using the union datatype defined in the prelude, but allows for any number of arguments, and doesn’t generate a datatype theory.

Typechecking expressions such as \(\text{IN}_1(3)\) requires that the context of its use be known. This is similar to the problem of a standalone \(\text{PROJ}_1\), and both are supported:

\[
\begin{align*}
\text{F}: & \ [\text{cT -> bool}] \\
\text{FF}: & \ \text{FORMULA } \text{F}(\text{IN}_1(3)) \\
\text{G}: & \ [\[\text{int -> [int, bool, [int -> int]]}] \rightarrow \text{bool} \\
\text{GG}: & \ \text{FORMULA } \text{G}(\text{PROJ}_1)
\end{align*}
\]

This means it is easy to write terms that are ambiguous:

\[
\begin{align*}
\text{HH}: & \ \text{FORMULA } \text{IN}_1(3) = \text{IN}_1(4) \\
\text{HH}: & \ \text{FORMULA } \text{PROJ}_1 = \text{PROJ}_1
\end{align*}
\]

This can be disambiguated by providing the type explicitly:

\[
\begin{align*}
\text{HH}: & \ \text{FORMULA } \text{IN}_1[\text{cT}](3) = \text{IN}_1(4) \\
\text{HH}: & \ \text{FORMULA } \text{PROJ}_1 = \text{PROJ}_1[[\text{int, int}]]
\end{align*}
\]

This uses the same syntax as for actual parameters, but doesn’t mean the same thing, as the projections, injections, etc., are builtin, and not provided by any theories. Note that coercions don’t work in this case, as \(\text{PROJ}_1::[[\text{int, int}] -> \text{int}]\) is the same as \((\text{LAMBDA} \ (x: [[\text{int, int}] -> \text{int}]): x)(\text{PROJ}_1)\)

and not \(\text{LAMBDA} \ (x: [\text{int, int}]): \text{PROJ}_1(x)\)

Note that the prover handles cotuple extensionality and reduction rules as expected.
4.5 Record Types

Record types are of the form \[# a_1:t_1, \ldots, a_n:t_n \#\]. The \(a_i\) are called *record accessors* or fields and the \(t_i\) are types. Record types are similar to tuple types, except that the order is unimportant and accessors are used instead of projections. Record types are empty if any of the component types is empty.

Note that the fields of a record type must be applied, they are not understood as functions. See Section 5.11.

4.6 Dependent types

Function, tuple, and record types may be dependent; in other words, some of the type components may depend on earlier components. Here are some examples:

\[
\text{rem: } [\text{nat}, d: \{n: \text{nat} \mid n /= 0\} \to \{r: \text{nat} \mid r < d\}] \\
\text{pfn: } [d: \text{pred}[\text{dom}], [[(d) \to \text{ran}]]] \\
\text{stack: } [# \text{size: nat, elements: } [\{n: \text{nat} \mid n < \text{size}\} \to t] \#]
\]

The declaration for \text{rem} indicates explicitly the range of the remainder function, which depends on the second argument. Function types may also have dependencies within the domain types; *e.g.*, the second domain type may depend on the first. Note that for function and tuple dependent types, local identifiers need to be given only for those types on which later types depend.

The tuple type \text{pfn} encodes partial functions as pairs consisting of a predicate on the domain type and a function from the subtype defined by that predicate to the range \text{ran}. If the second component were given instead as a function of type \([\text{dom} \to \text{ran}]\), then equality no longer works as intended. For example, the absolute value function \text{abs} and the identity function \text{id} are the same on the domain \text{nat}, so we would like to have

\[((\text{LAMBDA} (x:\text{int}):x >= 0), \text{abs}) = ((\text{LAMBDA} (x:\text{int}):x >= 0), \text{id})\]

but without the dependency this would be equivalent to \text{abs} = \text{id}.

\text{stack} encodes a stack as a pair consisting of a size and an array mapping initial segments of the natural numbers to \(t\). This is similar to the \text{pfn} example—in fact, if we were willing to use a tuple instead of a record encoding, \text{stack} could be declared as an instance of the type of \text{pfn}.

Another example, presented in [4] as a “challenge” to specification languages without partial functions, is easily handled with dependent types as shown below.

\[
\text{subp}(i:\text{int}, (j:\text{int} \mid i >= j)): \text{RECURSIVE int} = \\
\quad \text{(IF } i=j \text{ THEN } 0 \text{ ELSE } \text{subp}(i, j+1)+1 \text{ ENDIF)} \\
\quad \text{MEASURE } i - j
\]

However, some formulas that are valid with partial functions are not even well-formed in PVS:

\[
\text{subp_lemma: } \text{LEMMA } \text{subp}(i, 0) = i \text{ OR } \text{subp}(0, i) = i
\]

This generates unprovable \text{TCC}s. In practice this is rarely a problem.
4.7 Record and Tuple Type Extensions

Record and tuple types may now be extended using the `WITH` keyword. Thus, one may create colored points and moving points from simple points as follows.

- `point: TYPE = [# x, y: real #]`
- `colored_point: TYPE = point WITH [# color: Color #]`
- `moving_point: TYPE = point WITH [# vx, vy: real #]`

Similarly, tuples may be extended:

- `R3: TYPE = [real, real, real]`
- `R5: TYPE = R3 WITH [real, real]`

For record types, it is an error to extend with new field names that match any field names in the base record type. The extensions may not be dependent on the base type, though they may introduce dependencies within themselves.

- `dep_bad: TYPE = point WITH [# z: r: real | x*x + y*y < 1 #]`
- `dep_ok: TYPE = point WITH [# a: int, b: below(a) #]`

Note that the extension is a type expression, and may appear anywhere that a type is allowed.
4.7 Record and Tuple Type Extensions
Chapter 5

Expressions

The PVS language offers the usual panoply of expression constructs, including logical and arithmetic operators, quantifiers, lambda abstractions, function application, tuples, a polymorphic IF-THEN-ELSE, and function and record overrides. Expressions may appear in the body of a formula or constant declaration, as the predicate of a subtype, or as an actual parameter of a theory instance. The syntax for PVS expressions is shown in Figures 5.1 and 5.2.

The language has a number of predefined operators (although not all of these have a predefined meaning). These are given in Figure 5.3 below, along with their relative precedence from lowest to highest. Most of these operators are described in the following sections. IN is a part of LET expressions, WITH goes with override expressions, and the double colon (::) is for coercion expressions. The o operator is defined in the prelude as the function composition operator. Note that most operators may be overloaded, see Chapter 2 (page 7) for details.

Many of the operators may be overloaded by the user and retain their precedence and form (e.g., infix). All of the infix operators may also be given in prefix form; x + 1 and +(x,1) are semantically equivalent. Care must be taken in redefining these operators—if the preceding declaration ends in an expression there could be an ambiguity. To handle this situation the language allows declarations to be terminated with a ’;’. For example,

\[
\begin{align*}
\text{AND: } & [\text{state, state }\rightarrow\text{ state} ] = (\text{LAMBDA a,b: (LAMBDA t: a(t) AND b(t)))}; \\
\text{OR: } & [\text{state, state }\rightarrow\text{ state} ] = (\text{LAMBDA a,b: (LAMBDA t: a(t) OR b(t)))};
\end{align*}
\]

without the semicolon the second declaration would be seen as an infix OR and the result would be a parse error.

Another common mistake when overloading operators with predefined meanings is the assumption that overloading, for example, IMPLIES automatically provides an overloading for =>. This is not the case—they are distinct operators (which happen to have the same meaning by default) and not syntactic sugar.
### 5.1 Boolean Expressions

The Boolean expressions include the constants `TRUE` and `FALSE`, the unary operator `NOT`, and the binary operators `AND` (also written `&`), `OR`, `IMPLIES (=)`, `WHEN`, and `IFF (<=)`). The declarations for these are in the `booleans` prelude theory. All of these have their standard meaning, except for `WHEN`, which is the converse of `IMPLIES` (`i.e., A WHEN B ≡ B IMPLIES A`).

Equality (`=`) and disequality (`/=`) are declared in the prelude theories `equalities` and `notequal`. They are both polymorphic, the type depending on the types of the left- and right-hand sides. If the types are compatible, meaning that there is a common supertype, then the (dis)equality is of the greatest common supertype. Otherwise it is a type error. For example,

| S,T: TYPE  |
| s: VAR S  |
| t: VAR T  |
| eq1: FORMULA s = t |
| i: VAR {x: int | x < 10} |
| j: VAR {x: int | x > 100} |
| eq2: FORMULA i = j |

#### Figure 5.1: Expression syntax

| Expr ::= Number |
| String |
| Name |
| Id ! Number |
| Expr Arguments |
| Expr Binop Expr |
| Unaryop Expr |
| Expr \{ Id | Number \} |
| ( Expr^+ ) |
| ( Expr^+ :) |
| [ Expr^+ ] |
| { Expr^+ } |
| (# Assignment$^+$ #) |
| Expr : TypeExpr |
| IfExpr |
| BindingExpr |
| { SetBindings | Expr } |
| LET LetBinding$^+$ IN Expr |
| Expr WHERE LetBinding$^+$ |
| Expr WITH [ Assignment$^+$ ] |
| CASES Expr OF Selection$^+$ [ ELSE Expr ] ENDCASES |
| COND { Expr -> Expr$^+$ [ , ELSE -> Expr$^+$ ] ENDCOND |
| TableExpr |

50
Expressions

5.1 Boolean Expressions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>IfExpr</td>
<td>IF Expr THEN Expr { ELSIF Expr THEN Expr } * ELSE Expr ENDIF</td>
</tr>
<tr>
<td>BindingExpr</td>
<td>BindingOp LambdaBindings : Expr</td>
</tr>
<tr>
<td>BindingOp</td>
<td>LAMBDA</td>
</tr>
<tr>
<td>LambdaBindings</td>
<td>LambdaBinding [ , ] LambdaBindings</td>
</tr>
<tr>
<td>LambdaBinding</td>
<td>IdOp</td>
</tr>
<tr>
<td>SetBindings</td>
<td>SetBinding [ , ] SetBindings</td>
</tr>
<tr>
<td>SetBinding</td>
<td>(IdOp [ : TypeExpr ] )</td>
</tr>
<tr>
<td>Assignment</td>
<td>AssignArgs { :=</td>
</tr>
<tr>
<td>AssignArgs</td>
<td>Id [ ! Number ]</td>
</tr>
<tr>
<td></td>
<td>Number</td>
</tr>
<tr>
<td></td>
<td>AssignArg</td>
</tr>
<tr>
<td>AssignArg</td>
<td>( Expr )</td>
</tr>
<tr>
<td></td>
<td>` Id</td>
</tr>
<tr>
<td></td>
<td>` Number</td>
</tr>
<tr>
<td>Selection</td>
<td>IdOp [ ( IdOps ) ] : Expr</td>
</tr>
<tr>
<td>TableExpr</td>
<td>TABLE [ Expr ] [ , Expr ]</td>
</tr>
<tr>
<td></td>
<td>[ ColHeading ]</td>
</tr>
<tr>
<td></td>
<td>TableEntry ENDTABLE</td>
</tr>
<tr>
<td>ColHeading</td>
<td>[ { Expr }</td>
</tr>
<tr>
<td>TableEntry</td>
<td>{</td>
</tr>
<tr>
<td>LetBinding</td>
<td>{ LetBind</td>
</tr>
<tr>
<td>LetBind</td>
<td>IdOp Bindings* [ : TypeExpr ]</td>
</tr>
<tr>
<td>Arguments</td>
<td>( Expr )</td>
</tr>
</tbody>
</table>

Figure 5.2: Expression syntax (continued)

eq1 will cause a type error—remember that S and T are assumed to be disjoint. eq2 is perfectly typesafe because they have a common supertype int even though the subtypes have no elements in common; the equality simply has the value FALSE.

When the equality is between terms of type bool, the semantics are the same as for IFF. There is a pragmatic difference in the way the PVS prover processes these operators. Equalities may be used for rewriting, which makes for efficient proofs but is incomplete, i.e., the prover may fail to find the proof of a true formula. On the other hand the IFF form is complete, but may lead to a large number of cases. When in doubt, use equality as the
5.2 IF-THEN-ELSE Expressions

The IF-THEN-ELSE expression IF \( cond \) THEN \( expr1 \) ELSE \( expr2 \) ENDIF is polymorphic; its type is the common type of \( expr1 \) and \( expr2 \). The \( cond \) must be of type boolean. Note that the ELSE part is not optional as this is an expression, not an operational statement. The declaration for IF is in the if_def prelude theory. IF-THEN-ELSE may be redeclared by the user in the same way as AND, OR, etc. Note that only IF is explicitly redeclared, the THEN and ELSE are implicit.

Any number of ELSIF clauses may be present; they are translated into nested IF-THEN-ELSE expressions. Thus the expression
\[
\text{IF A THEN B}\n\text{ELSIF C THEN D}\n\text{ELSE E}\n\text{ENDIF}\]
translates to
\[
\text{IF A THEN B}\n\]

The prover provides commands that turn an equality into an IFF.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORALL, EXISTS, LAMBDA, IN</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-=</td>
</tr>
<tr>
<td>IFF, &lt;=&gt;</td>
<td>Right</td>
</tr>
<tr>
<td>IMPLIES, =&gt;, WHEN</td>
<td>Right</td>
</tr>
<tr>
<td>OR, //, XOR, ORELSE</td>
<td>Right</td>
</tr>
<tr>
<td>AND, &amp;, &amp;&amp;, /, ANDTHEN</td>
<td>Right</td>
</tr>
<tr>
<td>NOT, ~</td>
<td>None</td>
</tr>
<tr>
<td>=, /=, ==, &lt;, &lt;=, &gt;, &gt;=, &lt;=, &gt;, &lt;=, &gt;=, &lt;,</td>
<td>&gt;</td>
</tr>
<tr>
<td>WITH</td>
<td>Left</td>
</tr>
<tr>
<td>WHERE</td>
<td>Left</td>
</tr>
<tr>
<td>@, #</td>
<td>Left</td>
</tr>
<tr>
<td>@@, ##,</td>
<td></td>
</tr>
<tr>
<td>+, -, ++,</td>
<td>Left</td>
</tr>
<tr>
<td>*, /, **, //</td>
<td>Left</td>
</tr>
<tr>
<td>-</td>
<td>None</td>
</tr>
<tr>
<td>o</td>
<td>Left</td>
</tr>
<tr>
<td>:, ::, HAS_TYPE</td>
<td>Left</td>
</tr>
<tr>
<td>[], &lt;&gt;</td>
<td>None</td>
</tr>
<tr>
<td>^, ^^</td>
<td>Left</td>
</tr>
<tr>
<td>.</td>
<td>Left</td>
</tr>
</tbody>
</table>

Figure 5.3: Precedence Table
 ELSE (IF C THEN D
        ELSE E
        ENDIF)
ENDIF

5.3 Numeric Expressions

The numeric expressions include the numerals (0, 1, 2, ...), the unary operator -, and the binary infix operators ^, +, -, *, and /. The numerals are all of type real. The typechecker has implicit judgements on numbers; 0 is known to be real, rat, int and nat; all others are known to be non zero and greater than zero. The relational operators on numeric types are <, <=, >, and >=. The numeric operators and axioms are all defined in the prelude. As with the boolean operators, all of these operators may be defined on new types and retain their original precedences.

The numerals may also be treated as names, and overloaded. This is particularly useful for defining algebraic structures such as groups and rings, where it is natural to overload ‘0’ and ‘1’. Note that such use may include actual parameters, just as for names. Thus groups[int].0 or 0[int] might refer to the group zero instantiated with the integer carrier set.

5.4 Characters and String Expressions

String expressions are expressions enclosed in double quotes "", for example, "This is a string". Strings consist of eight bit ASCII characters. To include control characters or characters above the usual seven bits, use a back slash \, as described in the following table.

\a  ^G (BEL)
\b  ^H (backspace)
\f  ^L (form feed)
\n  ^J (new line)
\r  ^M (carriage return)
\t  ^I (horizontal tab)
\v  ^K (vertical tab)
\"  double quote
\\  backslash
\xNN  byte with hexadecimal value NN (2 digits)
\NNN  byte with decimal value NNN (3 digits)
\0NNN  byte with octal value NNN (3 digits)

Strings are finite sequences of characters, which in turn are represented by a datatype.

character: DATATYPE
BEGIN
char(code:below[256]):char?
When a string is parsed, it is internally converted to a conversion of a list of characters to a finite sequence. The following lemma is thus trivially true, because both sides are actually the same term.

```
string_rep: LEMMA
  "foo" = list2finseq(cons(char(102),
                 cons(char(111),
                 cons(char(111), null))))
```

Note that there is no special notation for characters; this is because the extract conversion will automatically convert a string of length one to a character. Note also that because of the finseq_convs conversion, a specific character may be extracted from a string simply by applying it. For example the following will typecheck

```
f: character = "f"
char_test: LEMMA "foo"(0) = f
```

## 5.5 Applications

Function application is specified as in ordinary mathematics; thus the application of function \( f \) to expression \( x \) is denoted \( f(x) \). Those operator symbols that are binary functions, and their applications, may be written in prefix or the usual infix notation. For example, \((3 + 5) = (2 * 4)\) may be written as \(+(3, 5), *(2, 4)\).

PVS supports higher-order types, so that functions may yield functions as values or be curried. For example, given \( f \) of type \( \mathrm{[int -> [int, int -> int]} \), \( f(0)(2,3) \) yields an \( \mathrm{int} \).

If the application involves a dependent function type then the result type of the application is substituted for accordingly. For example,

```
f: \{a:int, b:{x:int | a < x} -> {y:int | a < y & y <= b}\}
the application \( f(2,3) \) is of type \( \{y:int | 2 < y & y <= 3\} \). This application will also lead to the subtype \( \mathrm{TCC \ 2 < 3} \).
```

Application and tuple expressions have a special relation, due to the type equivalence of \( \mathrm{[t_1,\ldots,t_n -> t]} \) and \( \mathrm{[[t_1,\ldots,t_n] -> t]} \), see Section 5.9 for details.

## 5.6 Binding Expressions

The binding expressions are those which create a local scope for variables, including the quantified expressions and \( \lambda \)-expressions. Binding expressions consist of an operator, a list of bindings, and an expression. The operator is one of the keywords FORALL, EXISTS, or LAMBDA.\(^1\) The bindings specify the variables bound by the operator; each variable has an id and may also include a type or a constraint. Here is a contrived example:

```
x,y,z,d,e: \mathrm{VAR} \ real
ex1: AXIOM FORALL x,y,z: (x + y) + z = x + (y + z)
ex2: AXIOM FORALL (x,y,z: nat): x * (y + z) = (x * y) + (x * z)
ex3: AXIOM FORALL (n: num | n /= 0): EXISTS (x | x /= 0): x = 1/n
```

\(^1\)Set expressions are also binding expressions; see Section 5.8 (page 56).
In ex1, variables $x$, $y$, and $z$ are all of type $\text{real}$. In ex2 these same variables are of type $\text{nat}$, shadowing the global declarations. ex3 illustrates the use of constraints; this is equivalent to the declaration

\[
\text{ex3: AXIOM FORALL } \{n: \{n: \text{num} | n /= 0\}\}:
\exists \{x: \{x | x /= 0\}\}: x = 1/n
\]

Quantified expressions are introduced with the keywords FORALL and EXISTS. These expressions are of type boolean.

Lambda expressions denote unnamed functions. For example, the function which adds 3 to an integer may be written

\[
(\text{LAMBDA } (x: \text{int}): x + 3)
\]

The type of this expression is the function type $[\text{int -> numfield}].^2$ In addition, when the range is bool, a lambda expression may be represented as a set expression; see Section 5.8.

All of the binding expressions may involve dependent types in the bindings, e.g.,

\[
\text{FORALL } \{x: \text{int}\}, \{y: \{z: \text{int} | x < z\}\}: p(x, y)
\]

Note that in the instantiation of such an expression during a proof will generally lead to a subtype TCC. For example, substituting $e_1$ for $x$ and $e_2$ for $y$ will lead to the TCC $e_1 < e_2.$^3

Constant names may be treated as binding expressions by using a $!$ suffix. For example,

\[
\text{foo!} (x: \text{int}) : e
\]

is equivalent to

\[
\text{foo( LAMBDA } (x: \text{int}) : e)
\]

### 5.7 LET and WHERE Expressions

LET and WHERE expressions are provided for convenience, making some forms easier to read. Both of these forms provide local bindings for variables that may then be referenced in the body of the expression, thus reducing redundancy and allowing names to be provided for common subterms. Here are two examples:

\[
\text{LET } x: \text{int} = 2, y: \text{int} = x * x \text{ IN } x + y
\]

$x + y$ WHERE $x: \text{int} = 2, y: \text{int} = x * x$

The value of each of these expressions is 6.

LET and WHERE expressions are internally translated to applications of lambda expressions; in this case both expressions translate to

\[
(\text{LAMBDA } (x: \text{int}) : (\text{LAMBDA } (y: \text{int}) : x + y)(x * x))(2)
\]

These translations should be kept in mind when the semantics of these expressions is in question.

The type declaration is optional, so the above could be written as

\[
\text{LET } x = 2, y = x * x \text{ IN } x + y
\]

$x + y$ WHERE $x = 2, y = x * x$

In this case the typechecking of these expressions depends on whether $x$ and/or $y$ have been previously declared as variables. If they have, then those delarations are used to determine the type. Otherwise, the right-hand side of the $=$ is typechecked, and if it is unambiguous is

---

^2numfield sits between number and real, and is where the field operators are introduced. See Section prelude-numbers.

^3Such TCCs may never be seen, as they tend to be proved automatically during a proof; more complicated examples may be given, for which the prover would need help from the user. In addition, a false TCC can show up, e.g., substituting 2 for $x$ and 1 for $y$. This means that the corresponding expression is not type correct.
5.8 Set Expressions

In PVS, sets of elements of a type \( t \) are represented as predicates, i.e., functions from \( t \) to \( \text{bool} \). The type of a set may be given as \( [t \rightarrow \text{bool}] \), \( \text{pred}[t] \), or \( \text{setof}[t] \), which are all type equivalent. The choice depends wholly on the intended use of the type. Similarly, a set may be given in the form \( (\text{LAMBDA} \ (x: t): \ p(x)) \) or \( \{x: t | p(x)\} \); these are equivalent expressions. Note that the latter form may also represent a type—this usually causes no confusion as the context generally makes it clear which is expected. The usual functions and properties of sets are provided in the prelude theory \( \text{sets} \).

5.9 Tuple Expressions

A tuple expression of the type \( [t_1, \ldots, t_n] \) has the form \( (e_1, \ldots, e_n) \). For example, \( (2, \ \text{TRUE}, (\text{LAMBDA} \ (x: \ t): \ x + 1)) \) is of type \( [\text{nat}, \ \text{bool}, [\text{nat} \rightarrow \text{nat}]] \). 0-tuples are not allowed, and 1-tuples are treated simply as parenthesized expressions. The following relation holds between function types and tuple types:

\[
[[t_1, \ldots, t_n] \rightarrow t] \equiv [t_1, \ldots, t_n] \rightarrow t
\]

This equivalence is most important in theory parameters; it allows one theory to take the place of many. For example the functions theory from the prelude may be instantiated by the reference \( \text{injective?}[[[\text{int}, \text{int}, \text{int}]], \text{int}] \). Applications of an element \( f \) of this type include \( f(1,2,3), f((1,2,3)), \) and \( f(e) \), where \( e \) is of type \( [\text{int}, \text{int}, \text{int}] \).

5.10 Projection Expressions

The components of an expression whose type is a tuple can be accessed using the projection operators \( \ '1, \ '2, \ldots \) or \( \text{PROJ}_1, \text{PROJ}_2, \ldots \). The former are preferred. Like reserved words, projection expressions are case insensitive and may not be redeclared. For the most part, projection expressions are analogous to field accessors for record types. For example,

---

\(^4\)The prelude theory \texttt{defined_types} also defines \texttt{PRED}, \texttt{predicate}, \texttt{PREDICATE}, and \texttt{SETOF} as alternate equivalents.

\(^5\)In fact, internally they are represented by the same abstract syntax, they simply print differently.
t: [int, bool, [int -> int]]
ft: FORMULA t'2 AND t'1 > t'3(0)
ft_deprecated: FORMULA PROJ_2(t) AND PROJ_1(t) > (PROJ_3(t))(0)

Projection expressions may be used without an argument as long as the context determines the tuple type involved. For example, in the following it is obvious what tuple type is involved.

F: [[[int, bool, [int -> int]] -> bool] -> bool]
FP: FORMULA F(PROJ_2)

Note that the PROJ keyword must be used in such cases, as, e.g., `2 is not an expression. In the following example we see that the context does not provide enough information.

PP: FORMULA PROJ_2 = PROJ_2

To deal with such situations, the syntax for projections has been extended to allow the tuple type to be provided.

PP: FORMULA PROJ_2[[int, bool, [int -> int]]] = PROJ_2

In this case only one of the operators needs to be annotated. This looks like a use of actual parameters, but it is not, as the PROJ is not a name, and does not belong to a theory.

5.11 Record Expressions

Record expressions are of the form ( # a₁ := e₁, ..., aₙ := eₙ #), which has type [# a₁: t₁, ..., aₙ: tₙ #], where each eᵢ is of type tᵢ. Partial record expressions are not allowed; all fields must be given. If it is desired to give a partial record, declare an uninterpreted constant or variable of the record type, and use override expressions to specify the given record at the fields of interest. For example,

rc: [# a, b : int #]
re: [# a, b : int #] = rc WITH ['a := 0]

The type of a record expression is determined by the type of its components. Thus (# a := 3, b := 2 #) is of type [# a: real, b: real #]. This means that a record expression is never of a dependent record type directly, though it may be used where a dependent record is expected, and TCCs may be generated as a result. For example,

R: TYPE = [# a: int, b: {x: int | x < a} #]
r: R = (# a := 3, b := 4 #)

leads to the (unprovable) TCC 4 < 3.

Record expressions may be introduced without introducing the record type first, and the type of a record expression is determined by its components, independently of any previously declared record type. For this reason record types do not automatically generate associated accessor functions.

5.12 Record Accessors

The components of an expression of a record type are accessed using the corresponding field name. There are two forms of access. For example if r is of type [# x, y: real #], the x-component may be accessed using either r`x or x(r). The first form is preferred as there is less chance for ambiguity.

As noted above, accessors are not stand-alone functions. However, you can define your own functions to provide this capability, and even use the same name. For example:
5.13 Cotuple Expressions

Elements of cotuple types \([t_1 + \ldots + t_n]\) are constructed with the injection operators \(\text{IN}_i\) of type \([t_i \rightarrow [t_1 + \ldots + t_n]]\). Thus if \(e\) is of type \(t_i\), \(\text{IN}_i(e)\) is of the cotuple type. If \(x\) is an element of a cotuple type, \(\text{IN}_i?_i(x)\) is a boolean that tests if \(x\) belongs to the \(i^{th}\) component, and if it does, \(\text{OUT}_i?_i(x)\) returns the associated value of type \(t_i\). Note that this is similar to a datatype of the form

\[
\text{cotup}: \text{DATATYPE}
BEGIN
\text{IN}_1?_1(\text{OUT}_1: t_1): \text{IN}_1\_1
\ldots
\text{IN}_n?_n(\text{OUT}_n: t_n): \text{IN}_n\_n
END \text{cotup}
\]

The differences are that cotuples are not recursive, do not generate all the functions and axioms associated with datatypes, and allow for any number of component types—using datatypes a new one would have to be given for each arity.

The analogy works also for the \text{CASES} expression described in Section 9.4. This allows access to the values of a cotuple element. It has the form

\[
\text{CASES } e \text{ OF}
\begin{align*}
\text{IN}_1?_1(\text{OUT}_1: t_1): & f_1(x1), \\
\vdots \\
\text{IN}_n?_n(\text{OUT}_n: t_n): & f_n(xn)
\end{align*}
\text{ENDCASES}
\]

where each \(f_i\) is an expression of type \([t_i \rightarrow T]\), and the common return type \(T\) is the type of the \text{CASES} expression. For example, if \(x\) is of type \([\text{int + bool + [int -> int]}]\), the following expression will return a boolean value.

\[
\text{CASES } x \text{ OF}
\begin{align*}
\text{IN}_1?_1(\text{OUT}_1: t_1): & i > 0, \\
\text{IN}_2?_2(\text{OUT}_2: t_2): & \text{NOT } b, \\
\text{IN}_3?_3(\text{OUT}_3: t_3): & \text{FORALL } (n: \text{int}): f(f(n)) = f(n)
\end{align*}
\text{ENDCASES}
\]

If there are any missing components in the \text{CASES} expression, a \text{cases TCC} will be generated stating that the cotuple expression must be one of the given selections, unless there is an \text{ELSE} selection.

Like the projection operators \text{PROJ}_i, the \text{IN}_i, \text{OUT}_i and \text{IN}?_i operators make be disambiguated by adding the cotuple type reference to the operator, for example, \(\text{IN}_2?_2(\text{int + int})(3)\) or \(\text{IN}?_1?_1(\text{coT})\). Note that although they have the form of actual parameters, they are not, as these operators are built in and not associated with any theory. Also, for brevity, only the cotuple type is given, not the full type of the operator. There are a number of
Expressions

5.14 Override Expressions

Functions, tuples, records, and datatype elements may be “modified” by means of the override expression. The result of an override expression is a function, tuple, record, or datatype element that is exactly the same as the original, except that at the specified arguments it takes the new values. Keep in mind that there is no state change here, a new element is constructed from the existing one. For example,

\[ \text{id} \] \text{WITH} \ [(0) := 1, (1) := 2] \]

is the same function as the identity function (defined in the prelude) except at argument values 0 and 1. This is exactly the same expression as either of

\[ \text{(id} \text{WITH} \ [(0) := 1]) \text{WITH} \ [(1) := 2] \quad \text{or} \quad \text{(LAMBDA x: IF x = 1 THEN 2 ELSEIF x = 0 THEN 1 ELSE id(x))} \]

This order of evaluation ensures that functions remain total, and allows for the possibility of expressions such as

\[ \text{id} \text{WITH} \ [(c) := 1, (d) := 2] \]

where \( c \) and \( d \) may or may not be equal. If they are equal, then the value of the override expression at the common argument is 2.

More complex overrides can be made using nested arguments; for example,

\[
\begin{align*}
R : \text{TYPE} &= \text{[# a: int, b: [int -> [int, int]] #]} \\
r1 : R &= \text{R} \\
r2 : R &= r1 \text{WITH} \ [\text{`a := 0, `b(1)\text{\textquoteright}2 := 4}] \\
\end{align*}
\]

r2 is equivalent to

\[
\begin{align*}
(\# \text{ a := 0,} \\
\text{b := LAMBDA (x: int):} \\
\text{IF x = 1} \\
\text{THEN (r1\text{\textquoteright}b(x)\text{\textquoteright}1, 4)} \\
\text{ELSE r2\text{\textquoteright}b(x)} \\
\text{ENDIF #})
\end{align*}
\]

Updating a datatype element amounts to updating the accessor(s) associated with a constructor. For example, if \( \text{lst} \) is of type \( \text{cons?}\text{[nat]} \), then \( \text{lst} \text{WITH} \ [\text{\textquoteright}\text{car := 3}] \) returns a list that is the same as \( \text{lst} \), but whose first element is 3. If \( \text{lst} \) is given type \( \text{list}\text{[nat]} \), then the same override expression generates a TCC obligation to prove that \( \text{lst} \) is a \( \text{cons?} \). Because accessors may be both dependent and overloaded, TCCs may get complicated. For example,

\[

dt : \text{DATATYPE}
\begin{align*}
\text{BEGIN} \\
c0 : c0? \\
c1(x: \text{int, a: \{z: (even?) \mid z > x\}, b: \text{int}}): c1? \\
c2(x: \text{int, a: \{n: nat \mid n > x\}, c: \text{int}}): c2? \\
\text{END dt}
\end{align*}
\]

If \( d \) is of type \( dt \), the update expression \( d \text{ WITH} \ [\text{a := y}] \) leads to the TCC

\[
f1\_\text{TCC1: OBLIGATION} \\
(\text{c1?}(d) \text{ AND even?(y) AND } y > x(d)) \text{ OR} \\
(\text{c2?}(d) \text{ AND } y \geq 0 \text{ AND } y > x(d));
\]

Another form of override expression is the maplet, indicated using \( \mid -> \) in place of \( := \). This is used to extend the domain of the corresponding element; for example, if \( f: \text{[nat} \quad -> \text{nat]} \)
5.15 Coercion Expressions

Coercion expressions are of the form \( \text{expr} :: \text{type-expr} \), indicating that the expression \( \text{expr} \) is expected to be of type \( \text{type-expr} \). This serves two purposes. First, although PVS allows a liberal amount of overloading, it cannot always disambiguate things for itself, and coercion may be needed. For example, in

```plaintext
foo: int
foo: [int -> int]
foo: LEMMA foo = foo::int
```

the coercion of \( \text{foo} \) to \( \text{int} \) is needed, because otherwise the typechecker cannot determine the type. Note that only one of the sides of the equation needs to be disambiguated.

The second purpose of coercion is as an aid to typechecking; by providing the expected type in key places within complex expressions, the resulting TCCs may be considerably simplified.

5.16 Tables

Many expressions are easier to express and to read when presented in tabular form, as described in [7, 13]. There are many types of tables, ten different interpretations are described in [13] alone. Rather than provide support for all these tables, we chose to support a simple form of table initially, providing extensions in later versions of PVS as the need arises.
PVS provides a form of table expressions that allows simple tables\footnote{In Parnas’ terms [13], these tables are normal function tables of one or two dimensions.} to be presented, and supports table consistency conditions. One of the consistency conditions (the Mutual Exclusion Property or disjointness) requires the pairwise conjunction of a set of formulas to be false; another (the Coverage Property) requires the disjunction of a set of formulas to be true.

Tables are supported by means of the more generic COND expression, which provides the semantic foundation. In the following sections, we first describe the COND expression, and then TABLE expressions.

### 5.16.1 COND Expressions

The COND construct is a multi-way extension to the polymorphic IF-THEN-ELSE construct of PVS. Its form is

```plaintext
COND
  be_1 -> e_1,
  be_2 -> e_2,
  ...
  be_n -> e_n
ENDCOND
```

where the `be_i`'s are boolean expressions, and the `e_i`'s are expressions of some common supertype. It is required that the `be_i`'s are pairwise disjoint and that their disjunction is a tautology: these constraints are generated as disjointness and coverage TCCs that must be discharged before PVS will consider a COND expression fully type-correct.

```plaintext
foo_TCC1: OBLIGATION NOT (be_1 AND be_2) AND...AND NOT (be_n-1 AND be_n)
foo_TCC2: OBLIGATION be_1 OR be_2 OR...OR be_n
```

Notice that a COND expression with `n` clauses generates $O(n^2)$ clauses in its disjointness TCC.

Assuming its associated TCCs are discharged, the schematic COND shown above is equivalent to the following IF-THEN-ELSE form, which is its semantic definition.

```plaintext
IF be_1 THEN e_1
ELSIF be_2 THEN e_2
...
ELSIF be_n-1 THEN e_n-1
ELSE e_n
```

The COND may include an ELSE clause:

```plaintext
COND
  be_1 -> e_1,
  be_2 -> e_2,
  ...
  ELSE -> e_n
ENDCOND
```

This form does not require the coverage TCC and is equivalent to the IF-THEN-ELSE form shown above.
Using `COND`, we can translate the following tabular specification of the `sign` function

<table>
<thead>
<tr>
<th></th>
<th>$x &lt; 0$</th>
<th>$x = 0$</th>
<th>$x &gt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sign(x)</code></td>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

into

```plaintext
sign(x): int = COND
  x<0 -> -1,
  x=0 -> 0,
  x>0 -> 1
ENDCOND
```

Two dimensional tables can be generated by nested `CONDs`. For example, the following table defining the value for `safety_injection`

<table>
<thead>
<tr>
<th>modes</th>
<th>conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>false</td>
</tr>
<tr>
<td>low</td>
<td>not overridden</td>
</tr>
<tr>
<td>voter_failure</td>
<td>true</td>
</tr>
<tr>
<td>safety_injection</td>
<td>on</td>
</tr>
</tbody>
</table>

can be represented as

```plaintext
safety_injection(mode, overridden): on_off =
  COND
    mode=normal -> off,
    mode=low -> (COND NOT overridden -> on, overridden -> off ENDCOND),
    mode=voter_failure -> on
ENDCOND
```

Notice that `mode=low` provides the “left context” used in generating the TCCs for the nested `COND`. This causes some redundancy in highly structured two dimensional tables as the following example shows.

<table>
<thead>
<tr>
<th>state</th>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
</tbody>
</table>

This translates to

```plaintext
COND
  state=a -> COND input=x -> a,input=y -> b ENDCOND,
  state=b -> COND input=x -> b,input=y -> b ENDCOND
ENDCOND
```

The coverage TCCs generated for the two inner `CONDs` will have the form

```plaintext
foo_TCC2 : OBLIGATION state=a IMPLIES input=x OR input=y
foo_TCC3 : OBLIGATION state=b IMPLIES input=x OR input=y
```

whereas, because of the disjointness and coverage of `{a, b}`, the correct TCC is the simpler form

```plaintext
foo_TCC: OBLIGATION input=x OR input=y
```
The source of the error here is that our translation of the original table is too simple-minded. A better translation is the following.

\[
\text{LET } \begin{align*}
x1 &= \text{COND input=x \rightarrow a, input=y \rightarrow b ENDCOND}, \\
x2 &= \text{COND input=x \rightarrow b, input=y \rightarrow b ENDCOND} \\
\end{align*}
\text{IN} \\
\text{COND state=a \rightarrow x1, state=b \rightarrow x2 ENDCOND}
\]

And this generates the correct TCCs.

Note that if the be\_i’s are members of an enumerated type, then the standard PVS CASES construct should be used instead of COND, since there is no need to generate TCCs in these cases. For example, if in the previous example \{a, b\} and \{x, y\} had been enumerated types, then the table could have been expressed as

\[
\text{CASES state OF} \\
a: \text{CASES input OF} x: a, y: b ENDCASES, \\
b: \text{CASES input OF} x: b, y: b ENDCASES
\]

and no TCCs would be generated.

If the be\_i’s are all equalities with the same left hand side, whose right hand sides are ground arithmetic terms (involving only numbers, +, -, *, /) then the typechecker directly checks for coverage and disjointness so no TCCs are generated in this case.

### 5.16.2 Table Expressions

The COND and CASES constructs (see datatypes on page 81) provide the semantic foundation for our treatment of tables in PVS; for convenience, we also provide a TABLE construct that provides more attractive syntax for the important special cases of regular one and two-dimensional tables. The example above can be written in the alternative form.

\[
\text{TABLE} \\
\% --------------------- \\
| input=x | input=y |
\\%
\% ------------------------------- \\
| state=a | a | b ||
\%
\% ------------------------------- \\
| state=b | b | b ||
\%
\% ------------------------------- 
\%
\text{ENDTABLE}
\]

This will translate internally into the LET and COND form shown earlier. Note that the horizontal lines are simply PVS comments.\(^7\)

The row and column headers to a TABLE construct are arbitrary boolean expressions. In cases where the expressions are all of the form id=x, the id can be factored out to produce simpler tables of the following form.

\[
\text{TABLE state, input} \\
\% \\
\% \\
\% \\
\% -----------------
\%
\% | x | y | 
\%
\% | a | a | b |
\%
\% 
\%
\text{ENDTABLE}
\]

\(^7\)The \LaTeX{} generation translates these constructs into attractively typeset tables. See the PVS System Guide [11] for details.
In this form, as the headings are enumeration constructs this is internally represented as a CASES construct, and so generates no TCCs (the previous version generates 5 TCCs).

One-dimensional tables can be presented in both “horizontal” and “vertical” forms. The sign function example can be presented as a “vertical” table as follows.

```plaintext
sign(x): int = TABLE
% ------------------
| x<0 | -1 |
% ------------------
| x=0 |  0 |
% ------------------
| x>0 |  1 |
% ------------------
ENDTABLE
```

And as a horizontal one as follows.

```plaintext
sign(x): int = TABLE
% -------------------
| x<0 | x=0 | x>0 |
% -------------------
| -1  |  0  |  1  |
% -------------------
ENDTABLE
```

A more complex two-dimensional example is provided by the mode transition tables used in SCR. These have the following form.

<table>
<thead>
<tr>
<th>current mode</th>
<th>Event</th>
<th>New Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_1 )</td>
<td>( e_{1,1} )</td>
<td>( m_{1,1} )</td>
</tr>
<tr>
<td></td>
<td>( e_{1,2} )</td>
<td>( m_{1,2} )</td>
</tr>
<tr>
<td></td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td></td>
<td>( e_{1,k_1} )</td>
<td>( m_{1,k_1} )</td>
</tr>
<tr>
<td>( m_2 )</td>
<td>( e_{2,1} )</td>
<td>( m_{2,1} )</td>
</tr>
<tr>
<td></td>
<td>( e_{2,2} )</td>
<td>( m_{2,2} )</td>
</tr>
<tr>
<td></td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td></td>
<td>( e_{2,k_2} )</td>
<td>( m_{2,k_2} )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( m_p )</td>
<td>( e_{p,1} )</td>
<td>( m_{p,1} )</td>
</tr>
<tr>
<td></td>
<td>( e_{p,2} )</td>
<td>( m_{p,2} )</td>
</tr>
<tr>
<td></td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td></td>
<td>( e_{p,k_p} )</td>
<td>( m_{p,k_p} )</td>
</tr>
</tbody>
</table>

And translate to the following form.

```plaintext
TABLE mode
%--------------------------------------------------------------------------
```
The last row or column heading in a table may contain the **ELSE** keyword, which has the same meaning as for the corresponding **COND** or **CASES** expression.

The table may also have blank entries (except in the headings). These represent illegal values; in other words the entry may never be reached. This is represented by generation of a TCC indicating that the formulas corresponding to the row and column headings for that entry cannot both be true.

Note that this is different than having “don’t care” values. If you want to add don’t care entries, make sure that you use an array; the table

```
DC: int
TABLE
| x < 0 | x = 0 | x > 0 |
| y < 0 | 1 | 0 | DC |
| y = 0 | DC | 2 | 3 |
| y > 0 | -2 | DC | 0 |
ENDTABLE
```

may seem like any integer may appear in place of DC, but it must always be the same integer, which is probably not intended. The right way to do this is

```
DC(n:nat): int
TABLE
| x < 0 | x = 0 | x > 0 |
| y < 0 | 1 | 0 | DC(n) |
| y = 0 | DC(n) | 2 | 3 |
| y > 0 | -2 | DC(n) | 0 |
ENDTABLE
```
Chapter 6

Theories

Specifications in PVS are built from theories, which provide genericity, reusability, and structuring. PVS theories may be parameterized. A theory consists of a theory identifier, a list of formal parameters, an EXPORTING clause, an assuming part, a theory body, and an ending id. The syntax for theories is shown in Figure 6.1.

```
Specification ::= { Theory | RecursiveType } +
Theory ::= Id [ TheoryFormals ] : THEORY
          [ Exporting ]
          BEGIN
          [ AssumingPart ]
          [ TheoryPart ]
          END Id
TheoryFormals ::= [ TheoryFormal + ]
TheoryFormal ::= [ ( Importing ) ] TheoryFormalDecl
TheoryFormalDecl ::= Theory FormalType
                   | Theory FormalConst
                   | Theory FormalTheory
Theory FormalType ::= Ids : { TYPE | NONEMPTY_TYPE | TYPE+}
                   [ { FROM | <: } TypeExpr ]
Theory FormalConst ::= IdOps : TypeExpr
Theory FormalTheory ::= IdOps : THEORY TheoryDeclName
```

Figure 6.1: Theory Syntax

Everything is optional except the identifiers and the keywords. Thus the simplest theory has the form

```
triv : THEORY
BEGIN
END triv
```

The formal parameters, assuming, and theory body consist of declarations and importings. The various declarations are described in Section 3. In this section we discuss the restrictions on the allowable declarations within each section, the formal parameters, the assuming part, and the exportings and importings.
6.1 Theory Identifiers

The theory identifier introduces a name for a theory; as described in Section 8, this identifier can be used to help disambiguate references to declarations of the theory.

1Recall that o is an infix operator.
In the PVS system, the set of theories currently available to the session form a context. Within the context theory names must be unique. There is an initial context available, called the prelude that provides, among other things, the Boolean operators, equality, and the real, rational, integer, and natural number types and their associated properties. The only difference between the prelude and user-defined theories is that the prelude is automatically imported in every theory, without requiring an explicit IMPORTING clause.

The end identifier must match the theory identifier, or an error is signaled.

6.2 Theory Parameters

The theory parameters allow theory schemas to be specified. This provides support for universal polymorphism.

Theory parameters may be types, subtypes, constants, or theories, interspersed with importings. Theory parameters must have unique identifiers. The parameters are ordered, allowing later parameters to refer to earlier parameters or imported entities. This is another form of dependency, akin to dependent types (see Section 4.6). A theory is instantiated from within another theory by providing actual parameters to substitute for the formals. Actual parameters may occur in importings, exportings, theory declarations, and names. In each case they are enclosed in braces (i.e., { and }) and separated with commas.

The actuals must match the formals in number, kind, and (where applicable) type. In this matching process the importings, which must be enclosed in parentheses, are ignored. For example, given the theory declaration

\[
T \[t: \text{TYPE}, \\
\quad \text{subt: \text{TYPE FROM } t} \\
\quad \text{(IMPORTING orders[subt]) } \leq: \text{ (partial_order?)}, \\
\quad c: \text{subt}, \\
\quad d: \{x: \text{subt} \mid c \leq x\} \]
\]

a valid instance has five actual parameters; an example is

\[
T[int, \{x: \text{nat} \mid x < 10\}, <=, 5, 6] \]

Note that the matching process may lead to the generation of actual TCCs.

6.3 Importings and Exportings

The importing and exporting clauses form a hierarchy, much like the subroutine hierarchy of a programming language.

Names declared in a theory may be made available to other theories in the same context by means of the EXPORTING clause. Names exported by a given theory may be imported into a second theory by means of the IMPORTING clause. Names that are exported from one theory are said to be visible to any theory which uses the given theory. In this section we describe the syntax of the EXPORTING and IMPORTING clauses and give some simple examples.

6.3.1 The EXPORTING Clause

The EXPORTING clause specifies the names declared in the theory which are to be made available to any theory IMPORTING it. It may also specify instances of the theories which it imported to be exported. The syntax of the EXPORTING clause is given in Figure 6.5.

The EXPORTING clause is optional; if omitted, it defaults to

\[
\text{EXPORTING ALL WITH ALL} \]

\[2\]This is discussed in Chapter 7.
6.3 Importing and Exporting Theories

Any declared name may be exported except for variable declarations and formal parameters. When ALL is specified for the ExportingNames, all entities declared in the theory aside from the variables are exported. If a list of names is specified, then these are exported. Finally, when a list of names follows ALL BUT, all names aside from these are exported.

Since PVS supports overloading, it is possible that the exported name will be ambiguous. Such names may be disambiguated by including the type, if it is a constant, or by including one of the keywords TYPE or FORMULA. The keyword TYPE is used for any type declaration, and FORMULA is used for any formula declaration (including AXIOMs, LEMMAs, etc.) If not disambiguated, all declarations (except variables and formals) with the specified id will be exported.

When names are specified they are checked for completeness. This means that when a name is exported all of the names on which the corresponding declaration(s) depend must also be exported. Thus, for example, given the following declarations

```plaintext
sometype: TYPE
someconst: sometype
```

it would be illegal to export someconst without also exporting sometype. Note that this check is unnecessary if exporting ALL without the BUT keyword.

In some cases it is desirable (or necessary for completeness) to export some of the instances of the theories which are used by the given theory. This is done by specifying a WITH subclause as a part of the EXPORTING clause. The WITH subclause may be ALL, indicating that all instances of theories used by the given theory are exported. If CLOSURE is specified, then the typechecker determines the instances to be exported by a completion analysis on the exported names. Completion analysis determines those entities that are directly or indirectly referenced by one of the exported names.\(^3\) Finally, a list of theory names may be given; in this case the theory names must be complete in the sense that if an exported name refers to an entity in another theory instance, then that theory instance must be exported also. Other theory instances may also be exported even if not actually needed for completeness in this sense. The WITH subclause may only reference theory instances, i.e., theory names with actuals provided for all of the corresponding formal parameters.

As a practical matter, it is probably best not to include an EXPORTING clause unless there is a good reason. That way everything that is declared will be visible at higher levels of the IMPORTING chain.

### 6.3.2 IMPORTING Clauses

IMPORTING clauses import the visible names of another theory. IMPORTING clauses may appear in the formal parameters list, the assuming part, or the theory part of a theory. In addition, theory abbreviations implicitly

---

\(^3\)Proofs are not used in completion analysis.
import the theory name that they abbreviate (see Section 6.3.2).

The names appearing in an IMPORTING or theory abbreviation specifies a theory and optionally gives an instance of that theory, by providing actual parameters corresponding to the formal parameters of the theory used or mappings for the uninterpreted types and constants (see Chapter 7). IMPORTINGS are cumulative; entities made visible at some point in a theory are visible to every declaration following.

An IMPORTING with actual parameters provided is said to be a theory instance. We use the same terminology for an IMPORTING that refers to theory that has no formal parameters. Otherwise it is referred to as a generic reference.

A single theory may appear in any number of IMPORTINGs of another theory, both instantiated and generic. Obviously, any time there is more than one IMPORTING of a given theory there is a chance for ambiguity. Section 8 discusses such ambiguities, explaining how the system attempts to resolve them and how the user can disambiguate in situations where the system cannot.

An IMPORTING forms a relation between the theory containing the IMPORTING and the theory referenced. The transitive closure of the IMPORTING relation is called the importing chain of a theory. The importing chain must form a directed acyclic graph; hence a theory may not end up importing itself, directly or indirectly.

Theory Abbreviations
A theory abbreviation is a form of importing that introduces a new name for a theory instance, providing an alternate means for referring to the instance. For example, given the importing

\[
\text{IMPORTING sets[[integer -> integer]] AS fsets}
\]

where sets is a theory in which the function member is declared, the name sets[[integer -> integer]].member may instead be written as fsets.member.

6.4 Assuming Part

The assuming part consists of top-level declarations and IMPORTINGs. The assuming part precedes the theory part, so the theory part may refer to entities declared in the assuming part. The grammar for the assuming part is given in Figure 6.2.

The primary purpose of the assuming part is to provide constraints on the use of the theory, by means of ASSUMPTIONs. These are formulas expressing properties that are expected to hold of any instance of the theory. They are generally stated in terms of the formal parameters, and when instantiated they become assuming TCCs. For example, given the theory groups above, the importing

\[
\text{IMPORTING groups[int, 0, +, -]}
\]

generates the following obligations

\[
\text{IMP_groups_TCC1: OBLIGATION FORALL (a, b, c: int): a + (b + c) = (a + b) + c;}
\]

\[
\text{IMP_groups_TCC2: OBLIGATION FORALL (a: int): 0 + a = a AND a + 0 = a;}
\]

\[
\text{IMP_groups_TCC3: OBLIGATION FORALL (a: int): (\cdot)(a) + a = 0 AND a + (\cdot)(a) = 0;}
\]

Except for the variable declarations, the declarations of the assumings are all externally visible.

The dynamic semantics of an assuming part of a theory is as follows. Internal to the theory, assumptions are used exactly as axioms would be used. Externally, for each import of a theory, the assumptions have to be discharged (i.e., proved) with the actual parameters replacing the formal parameters. Note that in terms of the proof chain, every proof in a theory depends on the proofs of the assumptions.

Assuming TCCs are generated when a theory is instantiated, which may or may not occur when it is imported. Thus if a theory with assumptions is imported generically, the assuming TCCs are not generated until some reference is instantiated. If a theory instance is imported, then the assuming TCCs precede the importing in the dynamic semantics. Note that this may not make sense, as the assumings may refer to entities that are not visible until after the theory is imported. Thus the following is illegal.

\[4\]Prior to the introduction of theory interpretations, this was written as fsets: THEORY = sets[[integer -> integer]].
Assuming Part

assuming_test[n: nat, m: x:int | x < n]: THEORY
BEGIN
ASSUMING
rel_prime?(x, y: int): bool = EXISTS (a, b: int): x*a + y*b = 1
rel_prime: ASSUMPTION rel_prime?(n,m)
ENDASSUMING
END assuming_test

assimp: THEORY
BEGIN
IMPORTING assuming_test[4, 2]
END assimp

And leads to the following error message.

Error: assumption refers to assuming_test[4, 2].rel_prime?,
which is not visible in the current theory

There are a number of ways to solve this problem. Perhaps the simplest is to first import the theory generically, then import the instance.

IMPORTING assuming_test
IMPORTING assuming_test[4, 2]

Now the reference to rel_prime? makes sense in the assuming TCC generated for the second importing.

In this case, another solution is to simply define rel_prime? as a macro (see Section 3.5).

rel_prime?(x, y: int): MACRO bool = EXISTS (a, b: int): x*a + y*b = 1

Of course, this will not work if the declaration in question is a recursive or inductive definition.

Another solution is to provide the declaration in a theory that is imported in both the theory with the assuming and the theory importing that theory.

rel_prime[y:int]: THEORY
BEGIN
rel_prime?(x: int): bool = EXISTS (a, b: int): x*a + y*b = 1
END assth2

assuming_test[n: nat, m: x:int | x < n]: THEORY
BEGIN
ASSUMING
IMPORTING rel_prime[m]
rel_prime: ASSUMPTION rel_prime?(n)
ENDASSUMING
END assuming_test2

assuming_imp: THEORY
BEGIN
IMPORTING rel_prime[2], assuming_test[4, 2]
END assuming_imp

Now the reference to rel_prime? in the assuming TCC associated with assuming_test[4, 2] is the same as the previously imported instance, so there is no problem. In the theory assuming_imp, rel_prime may also be imported generically. However, if rel_prime is not imported, or is imported with a different parameter (e.g., rel_prime[3]) then the above error is produced.
6.5 Theory Part

The theory part consists of top-level declarations and IMPORTINGs. Declarations are ordered; references may not be made to declarations which occur later in the theory. The theory part usually contains the main body of the theory. Assuming declarations are not allowed in the theory part. The grammar for the theory part is given in Figure 6.3.
Chapter 7

Theory Interpretations

Theory interpretations are fully described in a technical report [10]. Here we give just the basics, for quick reference.

Theory interpretations in PVS follow the notion from logic, meaning that it allows uninterpreted types and constants of a given theory to be given interpretations. Such interpretations are given by including a mapping with a name reference. For example, here is one possible definition of a group theory:

```plaintext
group: THEORY
BEGIN
G: TYPE+
+: [G, G -> G]
0: G
-: [G -> G]
x, y, z: VAR G
associative_ax: AXIOM FORALL x, y, z: x + (y + z) = (x + y) + z
identity_ax: AXIOM FORALL x: x + 0 = x
inverse_ax: AXIOM FORALL x: x + -x = 0 AND -x + x = 0
idempotent_is_identity: LEMMA x + x = x => x = 0
END group
```

The uninterpreted types and constants of this theory are G, +, 0, and -. An example mapping for this is

```plaintext
group_inst: THEORY
BEGIN
IMPORTING group G := int, + := +, 0 := 0, - := - AS intG
realG: THEORY = group G := nzreal, + := *, 0 := 1, -(x: nzreal) := 1/x
END group_inst
```

This illustrates the usual ways theory instances are used: intG is a theory abbreviation, and realG is a theory declaration. The difference is that a theory abbreviation does not create new declarations; it is pretty much the same as treating the uninterpreted types and constants as formal parameters. On the other hand, realG is an inlined copy of the group theory, referenced as group_inst.realG. This leads to the possibility of nested theory declarations, and this is handled in PVS by allowing multiple dots, for example, group_inst.realG.inverse_ax.
Theory Interpretations

\[
\begin{align*}
\text{TheoryName} & ::= [ \text{Id} @] \text{Id} [\text{Actuals}] [\text{Mappings}] \\
\text{TheoryDeclName} & ::= [ \text{Id} @] \text{Id} [\text{Actuals}] [\text{TheoryMaps}] \\
\text{Name} & ::= [ \text{Id} @] \text{IdOp} [\text{Actuals}] [\text{Mappings}] [ . \text{IdOp}] \\
\text{Mappings} & ::= \{\{ \text{Mapping}^+ \}\} \\
\text{Mapping} & ::= \text{MappingLhs} \text{MappingRhs} \\
\text{MappingLhs} & ::= \text{IdOp Binding}^* [ : \{\text{TYPE} | \text{THEORY} | \text{TypeExpr} \}] \\
\text{MappingRhs} & ::= ::= \{\text{Expr} | \text{TypeExpr} \} \\
\text{TheoryMaps} & ::= \{\{ \text{TheoryMap}^+ \}\} \\
\text{TheoryMap} & ::= \text{MappingLhs} \text{TheoryMapRhs} \\
\text{TheoryMapRhs} & ::= \text{MapSubst} | \text{MapDef} | \text{MapRename} \\
\text{MapSubst} & ::= ::= \{\text{Expr} | \text{TypeExpr} \} \\
\text{MapDef} & ::= ::= \{\text{Expr} | \text{TypeExpr} \} \\
\text{MapRename} & ::= ::= \{\text{IdOp} | \text{Number} \}
\end{align*}
\]

Figure 7.1: Interpretation Syntax
Chapter 8

Name Resolution

Names in PVS are used to denote theories, variables, constants, and formulas. New names are introduced by declarations. The syntax of names is given in Figure 8.1.

The simplest form of a name is an idop, i.e., an identifier or operator symbol. This is generally all that is needed, unless names are overloaded.

The overloading of names, both from different theories and within a single theory, is allowed as long as there is some way for the system to distinguish references to them. Names from different theories may be distinguished by prefixing them with the theory name. Within a theory, all names of the same kind must be unique, except for expression kinds: which need only be unique up to the signature. This is because the signature is enough to distinguish these declarations. For example, if < is declared to have signature [bool, int -> bool], the system will recognize from the context that TRUE < 3 contains a reference to this declaration, whereas 2 < 3 does not.\(^1\) If the use of the name is not enough to distinguish, coercion may be used to specify the signature directly (see page 60). Theory parameters must be unique across all kinds.

There are three possible forms for names (two for theory names, which appear in IMPORTINGs, EXPORTING WITHs, and theory declarations). Given a theory named theoryid, with formal parameters \(f_1, \ldots, f_n\), that contains a declaration named id, the following three forms may be used to reference the declaration in a theory that imports theoryid:

- \(\text{theoryid}[a_1, \ldots, a_n].id\)
- \(id[a_1, \ldots, a_n]\)
- \(id\)

where the \(a_i\) are expressions or type expressions that are compatible with the formal parameters as described in Section 6.2. Note that any of these forms may have mappings immediately after the actual parameters. As described in Section 7, these can be viewed as an extension of the actuals. Note also that theory names allow different kinds of mappings. The forms above are listed in order of increasing likelihood of ambiguity—that is, names that are given with just an id are far more likely to produce an ambiguity than those further up. Note that even the top form may be ambiguous, as id may be declared more than once in theoryid. If this is the case, then either the context will disambiguate the name or a type will have to be supplied in the form of a coercion expression, e.g., id: : nat. This kind of ambiguity is allowed only for constants (including functions and recursive functions) and variables.

Names are resolved based on the expected type and the number and types of arguments to which the name is applied. The expected type is generally determined from the context of the name, for example in \(c_1: \text{int} = c_2\)

\(c_2\) has expected type \text{int}. For most expressions, this is straightforward, but applications create special problems. For example, in

\(^1\)Of course, this assumes that TRUE has not itself been overloaded.
we know that the equality (which is an application) has range type boolean since it is a formula, but this gives no information about the types of the arguments. We will first describe the simpler situation, and then explain how names used as operators of an application are resolved.

In general, the typechecker works by first collecting possible types for the expressions, and then chooses from among the possible types using the expected type, which is determined from the context of the expression. The expected type is used to resolve ambiguities, but otherwise does not contribute to the type of an expression. Thus if 2 + 3 typechecks, and + has not been redeclared, then it has type number_field regardless of its context. However, for the purpose of checking for TCCs, it may be treated as having a different type depending on the expected type and the available judgements.
### Name Resolution

The syntax for names and the structure of theories are defined as follows:

- **TheoryNames**: `TheoryName +
- **TheoryName**: `[Id] Id [Actuals] [Mappings]
- **TheoryDeclName**: `[Id] Id [Actuals] [TheoryMaps]
- **Names**: `Name +
- **Name**: `[Id] Id [Actuals] [Mappings] [ . IdOp]
- **Actuals**: `[Actual +]
- **Actual**: `Expr | TypeExpr
- **Mappings**: `{ Mapping + }
- **Mapping**: `MappingLhs MappingRhs
- **MappingLhs**: `IdOp Bindings + [ : { TYPE | THEORY | TypeExpr } ]
- **MappingRhs**: `:= { Expr | TypeExpr }
- **TheoryMaps**: `{ { TheoryMap + } }
- **TheoryMap**: `MappingLhs TheoryMapRhs
- **TheoryMapRhs**: `MapSubst | MapDef | MapRename
- **MapSubst**: `:= { Expr | TypeExpr }
- **MapDef**: `=: { Expr | TypeExpr }
- **MapRename**: `:= { IdOp | Number }
- **IdOps**: `IdOp +
- **IdOp**: `Id | Opsym | Number
- **Opsym**: `Binop | Unaryop | IF | TRUE | FALSE | [ || ] | ( || ) | { || }
- **Binop**: `o | IFF | <= > | IMPLIES | => | WHEN | OR | \| | AND
- **Unaryop**: `NOT | ~ | | [ ] | <> | -
- **FormulaName**: `AXIOM | CHALLENGE | CLAIM | CONJECTURE | COROLLARY
- **Fact**: `FORMULA | LAW | LEMMA | OBLIGATION
- **Postulate**: `PROPPOSITION | SUBLEMMA | THEOREM

Figure 8.1: Name Syntax
Chapter 9

Abstract Datatypes

PVS provides a powerful mechanism for defining abstract datatypes. This mechanism is akin to, but more sophisticated than, the shell principle of the Boyer-Moore prover [3]. A PVS datatype is specified by providing a set of constructors along with associated accessors and recognizers. When a datatype is typechecked, a new theory is created that provides the axioms and induction principles needed to ensure that the datatype is the initial algebra defined by the constructors.

The syntax for PVS datatypes is given in Figure 9.1. Datatypes may appear at the top-level as with theory declarations, or in-line as a declaration within a theory.\(^1\) Typechecking a top-level datatype named \texttt{foo} causes the generation of a new PVS file named \texttt{foo_adt.pvs} containing up to three theories as described below. Typechecking an in-line datatype has the effect of adding new declarations to the current theory, effectively replacing the in-line datatype. In-line datatypes are more restricted: they may not have formal parameters or assuming parts, and they will not generate the recursive combinators described below. The declarations

\[^1\text{Enumeration types are actually in-line datatypes—see Section 3.1.6.}\]
9.1 A Datatype Example: stack

An example of a datatype is stack:

```pvs
stack[T: TYPE]: DATATYPE
   BEGIN
      empty: empty?
      push(top:T, pop:stack): nonempty?
   END stack
```

The stack datatype has two constructors, empty and push, that allow stack elements to be constructed. For example, the term `push(1, empty)` is an element of type `stack[int]`. The recognizers `empty?` and `nonempty?` are predicates over the stack datatype that are true when their argument is constructed using the corresponding constructor. Given a stack element that is known to be `nonempty?`, the accessors `top` and `pop` may be used to extract the first and second arguments.

Typechecking the stack specification automatically creates a new file `stack_adt.pvs`, that contains the material found in the next five figures. This new file contains three theories: `stack_adt`, `stack_adt_map`, and `stack_adt_reduce`.

The first theory `stack_adt` is parametric in type T. This is a specification of “stacks of T”, where T may be instantiated by any defined type when the stacks datatype is imported. Thus “stacks of integers” as well as “stacks of stacks of integers” may be defined using this theory. The first few lines of the theory define the main type of stacks `stack`, the recognizers `emptystack?` and `nonemptystack?`, the constructors `empty` and `push`, and the accessors `top` and `pop` are declared.

The `stack_ord` function is defined, and an axiom provided for its definition. This is provided instead of a disjointness axiom, because the disjointness axiom becomes difficult to generate and use if the number of constructors is large. The disjointness comes from the fact that the natural numbers are distinct. The `ord` function is then defined to return 0 on an empty stack and 1 on a nonempty stack. This is the same function as `stack_ord`, but is easier to use.

Then a series of axioms are given. The `stack_empty_extensionality` axiom states that there is only one bottom element of the datatype: `empty`. The `stack_push_extensionality` states that any two stacks that have the same `top` and `pop` (have the same components) are the same. The `stack_push_eta` axiom states that popping and pushing the same element off and onto a stack results in a stack identical to the original. The `stack_top_push` says that if you push and element on a stack, you get that same element when you pop it back off. The `stack_pop_push` says that pushing something on a stack and then popping it back off results in the original stack.

The `stack_inclusive` axiom states that all stacks are either `empty?` or `nonempty?`. The PVS prover builds this axiom in, so that it rarely needs be cited by a user.
The next axiom, stack induction, introduces an induction formula for stacks stating that any predicate \( p \) of stacks that

1. holds for the empty stack (the base case), and
2. if \( p \) holds for some stack then \( p \) holds for the result of pushing anything of the right type onto that stack (the induction step),

then \( p \) holds for all stacks.

Then some useful functions are defined over stacks. The stack predicate every takes as arguments a predicate over \( T \) and a stack and returns TRUE iff all elements on the stack satisfy the given predicate. every is introduced in both curried and uncurried forms. The stack predicate some is dual to every, returning TRUE iff there is some element on the stack that satisfies the predicate. The subterm predicate takes two stacks and returns TRUE if and only if the first argument stack is a subterm of the second. That is, if the second stack
consists of the first stack with some (perhaps zero) elements pushed onto it. The \(<\leq\) predicate is the strict (irreflexive) subterm predicate. Thus for all stacks \(s\), \(\text{subterm}(s, s)\) holds, but for no stack \(s\) does \(<\leq(s, s)\) hold. An alternative equivalent definition of \(<\leq\) is as follows:
\[
<x: \text{stack}, y: \text{stack}>: \text{boolean} = \text{subterm}(x,y) \text{ AND NOT } x = y
\]
However, this definition is more awkward to use in a proof, as the recursion is hidden in the definition of subterm. For this reason the definitions for every, some, subterm, and \(<\leq\), are each defined as standalone functions, though some of them could be defined in terms of the others.

The last four declarations of the theory stack_adt are functions which reduce a stack to a natural number or to an ordinal. These functions are useful for simplifying the proof of termination of user-defined functions over stacks. Recall that PVS requires recursive functions to include a measure, which is used to generate termination conditions. The primary use of the recursive combinator is to allow measure functions to be specified. The function reduce_nat takes a natural number and a function. The natural number is used for the empty stack, and then for each element on the stack, the input function is applied to the element from the stack and the current reduced natural number, returning a natural number. The function reduce_nat returns the final natural number. The function REDUCE_nat is analogous to reduce_nat, except that the reducing function is also given the entire contents of the stack. This version of reduction can be useful for complicated measures that involve, for example, the number of repeated elements appearing on the stack. The simpler
9.1 A Datatype Example: stack

Form of reduce is difficult to apply to such situations. The functions reduce_ordinal and REDUCE_ordinal are analogous to reduce_nat and REDUCE_nat except that they return ordinal numbers instead of natural numbers. It is rare that a termination argument requires the use of ordinals, so the simpler reduce_nat form is more often used. This completes the description of the stack_adt theory.

The second theory in the file stack_adt.pvs is stack_adt_map. This theory takes two types T and T1 as parameters, imports the stack_adt theory, and defines a mapping from stacks[T] to stacks[T1]. The higher-order map function takes a function f of type [T -> T1], and a stack of T, and returns a stack of T1 obtained by applying f to each element on the input stack. map is defined in both curried and uncurried forms. map couldn’t reside in the stack_adt theory because that theory has only one type parameter, while the map functions require two: In order to construct and access stacks in two theories, map must be parameterized in the two types.

Also in the stack_adt_map is a relational every function. It lifts a relation R between T and T1, to stacks of T and T1. It is true if the stacks are the same size, and corresponding elements satisfy R.

The third and final theory generated from stack_pvs is stack_adt_reduce. This theory provides a
9.2 Datatype Details

In general, a datatype declaration has the form

```
adt: DATATYPE WITH SUBTYPES S₁, ..., Sₙ
   BEGIN
   cons₁(acc₁₁: T₁₁, ..., acc₁ₙ₁: T₁ₙ₁): rec₁ : S₁
   ....
   consₙ(accₙ₁: Tₙ₁, ..., accₙₙ: Tₙₙ): recₙ : Sₙ
   END adt
```

where the consᵢ are the constructors, the accᵢᵢ are the accessors, the Tᵢⱼ are type expressions, and the recᵢ are recognizers. Each line is referred to as a constructor specification. There are a number of restrictions enforced on constructor specifications.

The abstract datatype `REduce_ordinal` is defined as follows:

```
REduce_ordinal(empty?_fun: [stack -> ordinal],
                 nonempty?_fun: [[T, ordinal, stack] -> ordinal]):
  [stack -> ordinal] =
  LAMBDA (stack_adtvar: stack):
  LET red: [stack -> ordinal] = REduce_ordinal(empty?_fun, nonempty?_fun)
  IN
  CASES stack_adtvar
  OF empty: empty?_fun(stack_adtvar),
     push(push1_var, push2_var):
     nonempty?_fun(push1_var, red(push2_var), stack_adtvar)
  END CASES;
END stack_adt
```

The theory `stack_adt_map` is defined as follows:

```
stack_adt_map[T: TYPE, T1: TYPE]: THEORY
BEGIN
  IMPORTING stack_adt
  map(f: [T -> T1])(a: stack[T]): stack[T1] =
  CASES a
  OF empty: empty,
     push(push1_var, push2_var): push(f(push1_var), map(f(push2_var))
  END CASES;

  map(f: [T -> T1], a: stack[T]): stack[T1] =
  CASES a
  OF empty: empty,
     push(push1_var, push2_var): push(f(push1_var), map(f, push2_var))
  END CASES;

  every(R: [[T, T1] -> boolean])(x: stack[T], y: stack[T1]): boolean =
  empty?(x) AND empty?(y) OR
  nonempty?(x) AND
  nonempty?(y) AND R(top(x), top(y)) AND every(R)(pop(x), pop(y));
END stack_adt_map
```

This is a generalized version of `reduce_nat` and `REDUCE_nat`. It takes as parameters a type T and a range type range. It defines a generalized reduce which reduces stacks of T to elements of range. The functions `reduce_nat`, `REDUCE_nat`, `reduce_ordinal`, and `REDUCE_ordinal` could have been defined using `stack_adt_reduce`, but the direct definitions are provided for additional user convenience. The generalized reduce can be used to provide evidence of termination of user-defined functions, but the predefined versions such as `reduce_nat` are easier to use in most cases.
The datatype identifier may not be used for a recognizer, accessor, or subtype: (adt \neq \text{rec}_i \text{ for all } i, \text{adt} \neq \text{acc}_{ij} \text{ for all } i \text{ and } j, \text{ and } \text{adt} \neq S_i \text{ for all } i).

- The subtype names must be unique: \((i \neq j \Rightarrow S_i \neq S_j)\)

- Each subtype name must be used at least once.

- The constructor names must be unique: \((i \neq j \Rightarrow \text{cons}_i \neq \text{cons}_j)\).

- The recognizer names must be unique: \((i \neq j \Rightarrow \text{rec}_i \neq \text{rec}_j)\).

- No identifier may be used as both a constructor and a recognizer: \((\text{cons}_i, \neq \text{rec}_j \text{ for all } i \text{ and } j)\).

- Duplicate accessor identifiers are not allowed within a single constructor specification: \((j \neq k \Rightarrow \text{acc}_{ij} \neq \text{acc}_{ik})\).

As seen in the stack example, datatypes may be recursive; this is the case when the type of one or more of the accessors reference the datatype. In PVS, all such occurrences must be positive, where a type occurrence \(T\) is positive in a type expression \(\tau\) iff either

- \(\tau \equiv \text{T}\).

- \(\tau \equiv \{x : \tau' | p(x)\} \text{ and the occurrence } T \text{ is positive in } \tau'\).

- \(\tau \equiv [\tau_1 \rightarrow \tau_2] \text{ and the occurrence } T \text{ is positive in } \tau_2. \text{ For example, } T \text{ occurs positively in } \text{sequence}[T] \text{ where } \text{sequence}[T] \text{ is defined in the PVS prelude as the function type } [\text{nat} \rightarrow T]\).

- \(\tau \equiv [\tau_1, \ldots, \tau_n] \text{ and the occurrence } T \text{ is positive in some } \tau_i.\)

- \(\tau \equiv [\# l_1 : \tau_1, \ldots, l_n : \tau_n \#] \text{ and the occurrence } T \text{ is positive in some } \tau_i.\)

- \(\tau \equiv \text{datatype}[\tau_1, \ldots, \tau_n], \text{ where } \text{datatype} \text{ is a previously defined datatype and the occurrence } T \text{ is positive in } \tau_i, \text{ where } \tau_i \text{ is a positive parameter of } \text{datatype}.\)
When a top-level datatype is given with formal type parameters, they are checked for whether their occurrences are all positive; this is used as described above for any datatype that imports this one, as well as determining some of the declarations described below.

When a datatype is typechecked, a number of new declarations are generated:

- The datatype identifier is used to create an uninterpreted type declaration. In general, the term *datatype* refers to this type.
- Each recognizer is used to declare an uninterpreted subtype of the datatype.
- Each subtype identifier is used to declare an interpreted type that is the disjunction of the types given by the recognizers that reference the subtype identifier in the constructor specification.
- Each constructor and accessor is used to generate a constant declaration.
- An id ord uninterpreted function is created, and an axiom id ord_def axiom defines its values. This is provided instead of a disjointness axiom, because the disjointness axiom becomes difficult to generate and use when the number of constructors is large.
- An ord function is generated that gives a zero-based number to each constructor (e.g., ord(null) = 0 and cons(1,null) = 1). This is mostly useful for enumeration types.
- An extensionality axiom is generated for each constructor specification.
- An eta axiom is generated for each constructor specification that has accessors.
- For each accessor an axiom is created that says that the accessor composed with the corresponding constructor returns the correct value; e.g.,
  \[ \text{acc}_j(\text{cons}(\ldots, e_m)) = e_j \]
- An inclusive axiom is generated that says that every element of the datatype belongs to at least one recognizer subtype. This axiom is not actually needed in practice as the prover checks for this directly.
- Two induction schemes are provided for proving properties of the datatype.
- If there is at least one constructor with accessors, and there are positive type parameters to the datatype, then every and some functions are defined that provide a predicate on the datatype in terms of the positive types.
- The subterm and \(<\) (irreflexive subterm) functions are defined, and an axiom is generated that states that \(<\) is well-founded. This allows it to be used as an ordering relation in recursive function definitions.
- If there is at least one constructor with accessors, the reduce_nat, REDUCE_nat, reduce_ordinal, and REDUCE_ordinal recursion combinators are defined. These provide a means for defining notions like the size or depth of a datatype term.

Note that accessor subtypes involving the datatype are “lifted”. The following example shows why.

```
dt: DATATYPE
BEGIN
  c0: c0?
  c1(a1: {x: list[dt] | length(x) > 0}): c1?
  c2(a2: {x: list[dt] | every(c0?)(x)}): c2?
END dt
```

Consider the reduce_nat function. The signature for the lifted mapping function for c1 and c2 are the same: \([\text{list[nat]} \to \text{nat}]\). It’s obvious the mapping function for c2 function could have the signature \([x: \text{list[nat]} | \text{length}(x) > 0] \to \text{nat}\], but there is no obvious way to map c2 without lifting it. Since it is not trivial to determine which predicates map nicely, we lift them all. In the future we may provide heuristics that refine this.

\(^2\)Note that enumeration types have no accessors.
• If some type parameter is positive a map function is generated in a separate theory. Every positive
type parameter in the datatype is associated with a pair of map parameters, which form the domain and
range of a corresponding function argument. Given a set of such functions and a term of the datatype,
map returns a term that has the same structure, but with the “leaf” elements replaced by the function
values.
• A separate theory is generated for the reduce and REDUCE functions. These generalize the reduce
functions above to an arbitrary range type.

Note that in the stack example, the stack type is nonempty, since empty is an element of stack even if
the parameter type T is instantiated with an empty type. However, there is no requirement that a datatype be
nonempty, though if it is imported and a constant is declared to be of that type, a TCC will be generated as
described on page 15 in section 3.1.7.

The stack_adt theory is parameterized in the type T, and introduces the uninterpreted type stack. Under
normal circumstances, this would imply no relation between, for example, stack[nat] and stack[int].
However, since every occurrence of T in the accessor types is positive, we can infer that stack[nat] is a
subtype of stack[int]. In general, given a type T and a subtype \( S \equiv \{x : T | p(x)\} \), then stack[S] is
treated the same as \( \{s : \text{stack}[T] | \text{every}(p)(s)\} \). When a datatype has a mix of positive and nonpositive type
parameters, the subtype relation only holds for the positive ones. For example, in the datatype

```
dt[T1, T2: TYPE, c: T1]: DATATYPE
BEGIN
c(a1: T1, a2: [T2 -> T1]): c?
END dt
```

T1 is positive and T2 is not, so \( \text{dt}[\text{nat}, \text{nat}, 0] \) is a subtype of \( \text{dt}[\text{int}, \text{nat}, 0] \), but is not a subtype of
\( \text{dt}[\text{nat}, \text{int}, 0] \), nor is it a subtype of \( \text{dt}[\text{nat}, \text{nat}, 1] \).

More complex datatypes lead to correspondingly more complex declarations; for example, in the follow-
ing contrived datatype

```
adt1[t1,t2: TYPE, c:t1]: DATATYPE
BEGIN
bottom: bottom?
c1(a11:t1, a12: [t2 -> int]): c1?
c2(a21:adt1, a22: [nat -> adt1], a23: list[adt1]): c2?
c3(a31:[list[int] -> adt1],
    a32:[# a: adt1, b: [int -> adt1] #],
    a33: [adt1, [set[int] -> adt1]]): c3?
END adt1
```

the curried every is generated as follows:
9.3 Datatype Subtypes

The **WITH SUBTYPES** keyword introduces a set of subtype names. These are useful, for example, in defining the nonterminals of a language. For example, we might try to describe a simple typed lambda calculus:

\[
\begin{align*}
T & ::= B \mid T \to T \\
E & ::= x \mid \lambda x : T. E \mid E(E)
\end{align*}
\]

This is difficult to express using datatypes without subtypes, but is reasonably straightforward with them:

\[
\begin{align*}
tlc: & \text{DATATYPE WITH SUBTYPES typ, expr} \\
& \text{BEGIN} \\
& \text{base_type(n:nat): base_type? : typ} \\
& \text{fun_type(dom, ran: typ): fun_type? : typ} \\
& \text{expr_var(n:nat): expr_var? : expr} \\
& \text{lambda_expr(lvar:(expr_var?), ltype: typ, lexpr: expr)} \\
& \hspace{1cm} : \text{lambda_expr? : expr} \\
& \text{application(fun, arg: expr): application? : expr} \\
& \text{END tlc}
\end{align*}
\]

In addition to the usual generated declarations, this generates

\[
^3\text{TYPE, LAMBDA, and VAR are PVS keywords, so variants are used here.}
\]
immediately after the declarations generated for the recognizers, so they may be referenced in the accessor types. Note that only a single induction scheme is generated. To induct over a particular subtype, extend the property of interest to the entire datatype so that it returns true for everything else.

## 9.4 CASES Expressions

The CASES expression uses a simple form of pattern-matching on abstract datatypes. Patterns are of the form $c(x_1, \ldots, x_n)$ where $c$ is an $n$-ary constructor and $x_1, \ldots, x_n$ is a list of distinct variables. Patterns here are simple so that certain logical properties of the expression are easy to check. Patterns are not defined in the grammar but in the type rules, since the notion of a variable or a constructor is only defined in the type rules.

For example, if $x$ is of type stack, the cases expression

```
CASES x OF
  empty : FALSE,  
  push(y, z) : even?(y) AND empty?(z)
ENDCASES
```

is TRUE if $x$ is a singleton even integer, and otherwise is false. This expression can be translated into

```
IF empty?(x) THEN FALSE
ELSE LET (y, z) = (car(x), cdr(x))
  IN even?(y) AND empty?(z)
ENDIF
```

The CASES expression also allows an ELSE clause, which comes last and covers all constructors not previously mentioned in a pattern. If the ELSE clause is missing, and not all constructors have been mentioned, then a cases TCC is generated which states that the expression is not any of the missing elements. For example, if the $x$ above is declared to be a subtype of stack in which empty is excluded, then the empty case can safely be left out, and a TCC will be generated that obligates the user to prove that $x$ is not empty. There is a trade-off here between simpler specifications and simpler verifications; if the empty case is left in, then there is no obligation to prove, but the extra case clutters up the specification, and can mislead the reader into thinking that the empty case is possible. In general, we feel that the specification should be as perspicuous as possible, even if it means a little more work behind the scenes.
Appendix A

The Grammar

The complete PVS grammar is presented in this Appendix, along with a discussion of the notation used in presenting the grammar.

The conventions used in the presentation of the syntax are as follows.

- Names in *italics* indicate syntactic classes and metavariables ranging over syntactic classes.
- The reserved words of the language are printed in tt font, UPPERCASE.
- An optional part \( A \) of a clause is enclosed in square brackets: \([A]\).
- Alternatives in a syntax production are separated by a bar (“|”); a list of alternatives that is embedded in the right-hand side of a syntax production is enclosed in brackets, as in
  
  \[
  \text{ExportingName} \ := \ \text{IdOp} \ [ \ { \text{TypeExpr} \mid \text{TYPE} \mid \text{FORMULA} } ]
  \]
- Iteration of a clause \( B \) one or more times is indicated by enclosing it in brackets followed by a plus sign: \( B^+ \); repetition zero or more times is indicated by an asterisk instead of the plus sign: \( B^* \).
- A double plus or double asterisk indicates a clause separator; for example, \( B^+ \) indicates zero or more repetitions of the clause \( B \) separated by commas.
- Other items printed in tt font on the right hand side of productions are literals. Be careful to distinguish where BNF symbols occur as literals, e.g., the BNF brackets \{ \} versus the literal brackets \{ \}. 
The Grammar

Specification

\[ \text{Specification} ::=} \{ \text{Theory} | \text{RecursiveType} \}^+ \]

\[ \text{Theory} ::=} \text{Id} \{ \text{TheoryFormals} \} : \text{THEORY} \]

\[ \text{BEGIN} \]
\[ \text{AssumingPart} \]
\[ \text{TheoryPart} \]
\[ \text{END Id} \]

\[ \text{TheoryFormals} ::=} \{ \text{TheoryFormal}^+ \} \]

\[ \text{TheoryFormal} ::=} \{ \text{Importing} \} \text{TheoryFormalDecl} \]

\[ \text{TheoryFormalDecl} ::=} \text{TheoryFormalType} \]

\[ \text{TheoryFormalConst} ::=} \text{IdOps} : \text{TypeExpr} \]

\[ \text{TheoryFormalTheory} ::=} \text{IdOps} : \text{THEORY TheoryDeclName} \]

RecursiveTypes

\[ \text{RecursiveType} ::=} \text{Id} \{ \text{TheoryFormals} \} : \{ \text{DATATYPE} | \text{CODATATYPE} \} \{ \text{WITH SUBTYPES Ids} \}
\[ \text{BEGIN} \]
\[ \text{Importing [ ; ]} \]
\[ \text{AssumingPart} \]
\[ \text{RecursiveTypePart} \]
\[ \text{END Id} \]

\[ \text{InlineRecursiveType} ::=} \text{Id} \{ \text{DeclFormals} \} : \{ \text{DATATYPE} | \text{CODATATYPE} \} \{ \text{WITH SUBTYPES Ids} \}
\[ \text{BEGIN} \]
\[ \text{Importing [ ; ]} \]
\[ \text{AssumingPart} \]
\[ \text{DatatypePart} \]
\[ \text{END id} \]

\[ \text{RecursiveTypePart} ::=} \{ \text{Constructor} : \text{IdOp} [ : \text{Id} ] \}^+ \]

\[ \text{Constructor} ::=} \text{IdOp} \{ \{ \text{IdOps} : \text{TypeExpr} \}^+ \} \]
### Assumings

| AssumptionPart | ::= | ASSUMING \{AssumingElement ;\}^+ ENDASSUMING |
| AssumingElement | ::= | Importing |
| | | Decl |
| | | Assumption |
| Assumption | ::= | Ids [DeclFormals] : ASSUMPTION Expr |

### Theory Part

| TheoryPart | ::= | \{TheoryElement ;\}^+ |
| TheoryElement | ::= | Importing | Decl |
| Decl | ::= | LibDecl | TheoryDecl | TypeDecl | VarDecl |
| | | ConstDecl | RecursiveDecl | MacroDecl | InductiveDecl |
| | | ColInductiveDecl | FormulaDecl | Judgement | Conversion |
| | | InlineRecursiveType | AutoRewriteDecl |

### Importings and Exportings

| Exporting | ::= | EXPORTING ExportingNames [WITH ExportingTheories] |
| ExportingNames | ::= | ALL [ BUT ExportingName^+ ] |
| | | ExportingName^+ |
| ExportingName | ::= | PIdOp [: \{TypeExpr | TYPE | FORMULA\}] |
| ExportingTheories | ::= | ALL | CLOSURE | TheoryNames |
| Importing | ::= | IMPORTING ImportingItem^+ |
| ImportingItem | ::= | TheoryName [ AS Id] |
### Declarations

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>LibDecl</td>
<td>LibDecl ::= Ids : LIBRARY [ = ] String</td>
</tr>
<tr>
<td>TheoryDecl</td>
<td>TheoryDecl ::= Ids [ DeclFormals ] : THEORY = TheoryDeclName</td>
</tr>
<tr>
<td>TypeDecl</td>
<td>TypeDecl ::= Id [ DeclFormals ] [ / , Ids ]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>VarDecl</td>
<td>VarDecl ::= IdOps [ DeclFormals ] : VAR TypeExpr</td>
</tr>
<tr>
<td>ConstDecl</td>
<td>ConstDecl ::= IdOp [ DeclFormals ] [ / , IdOps ]</td>
</tr>
<tr>
<td>RecursiveDecl</td>
<td>RecursiveDecl ::= IdOp [ DeclFormals ] [ / , IdOps ]</td>
</tr>
<tr>
<td>MacroDecl</td>
<td>MacroDecl ::= IdOp [ DeclFormals ] [ / , IdOps ]</td>
</tr>
<tr>
<td>InductiveDecl</td>
<td>InductiveDecl ::= IdOp [ DeclFormals ] [ / , IdOps ]</td>
</tr>
<tr>
<td>ColInductiveDecl</td>
<td>ColInductiveDecl ::= IdOp [ DeclFormals ] [ / , IdOps ]</td>
</tr>
<tr>
<td>FormulaDecl</td>
<td>FormulaDecl ::= Ids [ DeclFormals ] : FormulaName Expr</td>
</tr>
<tr>
<td>Judgement</td>
<td>Judgement ::= SubtypeJudgement</td>
</tr>
<tr>
<td>SubtypeJudgement</td>
<td>SubtypeJudgement ::= [ IdOp [ DeclFormals ] : ] JUDGEMENT TypeExpr+ SUBTYPE_OF TypeExpr</td>
</tr>
<tr>
<td>ConstantJudgement</td>
<td>ConstantJudgement ::= [ IdOp [ DeclFormals ] : ] [ RECURSIVE JUDGEMENT ConstantReference+</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ConstantReference</td>
<td>ConstantReference ::= Name Bindings*</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Conversion</td>
<td>Conversion ::= { CONVERSION</td>
</tr>
<tr>
<td>AutoRewriteDecl</td>
<td>AutoRewriteDecl ::= { AUTO_REWRITE</td>
</tr>
<tr>
<td>RewriteName</td>
<td>RewriteName ::= Name [ ! [ ! ] ] [ : { TypeExpr</td>
</tr>
<tr>
<td>DeclFormal</td>
<td>DeclFormal ::= [ DeclFormal+ ]</td>
</tr>
<tr>
<td>DeclFormals</td>
<td>DeclFormals ::= TheoryFormalType</td>
</tr>
<tr>
<td>Bindings</td>
<td>Bindings ::= ( Binding+ )</td>
</tr>
<tr>
<td>Binding</td>
<td>Binding ::= TypedId</td>
</tr>
<tr>
<td>TypedIds</td>
<td>TypedIds ::= IdOps [ : TypeExpr ] [</td>
</tr>
<tr>
<td>TypedId</td>
<td>TypedId ::= IdOp [ : TypeExpr ] [</td>
</tr>
</tbody>
</table>
Type Expressions

\[
TypeExpr \ ::= \begin{array}{l}
TypeExpr \mid
\text{TypeName} \\
\text{EnumerationType} \\
\text{Subtype} \\
\text{TypeApplication} \\
\text{FunctionType} \\
\text{TupleType} \\
\text{CotupleType} \\
\text{RecordType}
\end{array}
\]

\[
\begin{align*}
\text{TypeName} & \ ::= \ \text{Name} \ [\text{WITH TypeExpr}] \\
\text{EnumerationType} & \ ::= \ \{ \text{IdOps} \} \\
\text{Subtype} & \ ::= \ \{ \ \text{SetBindings} \ | \ \text{Expr} \ \} \ [\text{WITH TypeExpr}] \\
& \quad | \ ( \ \text{Expr} \ ) \ [\text{WITH TypeExpr}] \\
\text{TypeApplication} & \ ::= \ \text{Name Arguments} \ [\text{WITH TypeExpr}] \\
\text{FunctionType} & \ ::= \ [\ \text{FUNCTION} \ | \ \text{ARRAY} \]
& \quad [\ \{ \ \text{IdOp} : \ \text{TypeExpr} \ \}^+ \ -\to \ \text{TypeExpr} \ ] \\
\text{TupleType} & \ ::= \ [\ \{ \ \text{IdOp} : \ \text{TypeExpr} \ \}^+ \ ] \ [\text{WITH TypeExpr}] \\
\text{CotupleType} & \ ::= \ [\ \{ \ \text{IdOp} : \ \text{TypeExpr} \ \}^+_\ -\to\ ] \\
\text{RecordType} & \ ::= \ [\ # \ \text{FieldDecls}^+_\ -\to \ #] \ [\text{WITH TypeExpr}] \\
\text{FieldDecls} & \ ::= \ \text{Ids} : \ \text{TypeExpr}
\end{align*}
\]
Expressions

\[
\text{Expr} ::= \text{Number} \\
| \text{String} \\
| \text{Name} \\
| \text{Id} \uparrow \text{Number} \\
| \text{Expr Arguments} \\
| \text{Expr Binop Expr} \\
| \text{Unaryop Expr} \\
| \text{Expr} ` \{ \text{Id} \mid \text{Number} \} \\
| (\text{Expr}^+) \\
| (\text{:} \text{Expr}^+:) \\
| [\text{Expr}^+] \\
| (\text{\mid} \text{Expr}^+) \\
| \{\text{Expr}^+\} \\
| (\# \text{Assignment}^+\#) \\
| \text{Expr} :: \text{TypeExpr} \\
| \text{IfExpr} \\
| \text{BindingExpr} \\
| \{ \text{SetBindings} \mid \text{Expr} \} \\
| \text{LET LetBinding}^+ \text{IN Expr} \\
| \text{Expr WHERE LetBinding}^+ \\
| \text{Expr WITH} [\text{Assignment}^+] \\
| \text{CASES Expr} \text{OF Selection}^+ \{\text{ELSE Expr}\} \text{ENDCASES} \\
| \text{COND} \{\text{Expr} -> \text{Expr}\}^+ [\text{ELSE -> Expr}] \text{ENDCOND} \\
| \text{TableExpr}
\]
Expressions (continued)

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>IfExpr</td>
<td>:= IF Expr THEN Expr { ELSIF Expr THEN Expr } ELSE Expr ENDIF</td>
</tr>
<tr>
<td>BindingExpr</td>
<td>:= BindingOp LambdaBindings : Expr</td>
</tr>
<tr>
<td>BindingOp</td>
<td>:= LAMBDA</td>
</tr>
<tr>
<td>LambdaBindings</td>
<td>:= LambdaBinding [ [, ] LambdaBindings ]</td>
</tr>
<tr>
<td>LambdaBinding</td>
<td>:= IdOp</td>
</tr>
<tr>
<td>SetBindings</td>
<td>:= SetBinding [ [, ] SetBindings ]</td>
</tr>
<tr>
<td>SetBinding</td>
<td>:= {IdOp [ : TypeExpr] }</td>
</tr>
<tr>
<td>Assignment</td>
<td>:= AssignArgs { :=</td>
</tr>
<tr>
<td>AssignArgs</td>
<td>:= Id [ ! Number ]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>AssignArg</td>
<td>:= ( Expr^ )</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td>:= IdOp [ ( IdOps ) ] : Expr</td>
</tr>
<tr>
<td>TableExpr</td>
<td>:= TABLE [ Expr] [ , Expr ] [ ColHeading ] TableEntry^ ENDTABLE</td>
</tr>
<tr>
<td>ColHeading</td>
<td>:=</td>
</tr>
<tr>
<td>TableEntry</td>
<td>:=</td>
</tr>
<tr>
<td>LetBinding</td>
<td>:= { LetBind</td>
</tr>
<tr>
<td>LetBind</td>
<td>:= IdOp Bindings^ [ : TypeExpr ]</td>
</tr>
<tr>
<td>Arguments</td>
<td>:= ( Expr^ )</td>
</tr>
</tbody>
</table>
The Grammar

Names

TheoryNames ::= TheoryName +
TheoryName ::= [Id @] Id [Actuals] [Mappings]
TheoryDeclName ::= [Id @] Id [Actuals] [TheoryMaps]
Names ::= Name +
Name ::= [Id @] IdOp [Actuals] [Mappings] [ . IdOp]
Actuals ::= [Actual +]
Actual ::= Expr | TypeExpr
Mappings ::= { { Mapping + } }
Mapping ::= MappingLhs MappingRhs
MappingLhs ::= IdOp Bindings + [ : { TYPE | THEORY | TypeExpr } ]
MappingRhs ::= := { Expr | TypeExpr }
TheoryMaps ::= { { TheoryMap + } }
TheoryMap ::= MappingLhs TheoryMapRhs
TheoryMapRhs ::= MapSubst | MapDef | MapRename
MapSubst ::= ::= { Expr | TypeExpr }
MapDef ::= = { Expr | TypeExpr }
MapRename ::= ::= { IdOp | Number }
IdOps ::= IdOp +
IdOp ::= Id | Opsym | Number
Opsym ::= Binop | Unaryop
Binop ::= | IF | TRUE | FALSE | [ | ] | ( | ) | { | } |
Unaryop ::= NOT | ~ | [ ] | <> | -

FormulaName ::= AXIOM | CHALLENGE | CLAIM | CONJECTURE | COROLLARY
FACT | FORMULA | LAW | LEMMA | OBLIGATION
POSTULATE | PROPOSITION | SUBLEMMA | THEOREM

100
**Identifiers**

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Ids</em></td>
<td><code>::= Id^+</code></td>
</tr>
<tr>
<td><em>Id</em></td>
<td><code>::= Letter IdChar^+</code></td>
</tr>
<tr>
<td><em>Number</em></td>
<td><code>::= Digit^+</code></td>
</tr>
<tr>
<td><em>String</em></td>
<td><code>::= &quot; Unicode-character* &quot;</code></td>
</tr>
<tr>
<td><em>IdChar</em></td>
<td>`::= Letter</td>
</tr>
<tr>
<td><em>Letter</em></td>
<td>`::= A</td>
</tr>
<tr>
<td><em>Digit</em></td>
<td>`::= 0</td>
</tr>
</tbody>
</table>
Bibliography


Index

*, 53
+, 53
-, 53
\texttt{.pvscontext}, 34
/, 53
/=, 50
<, 53
<=, 53
\texttt{\&}, 50
\texttt{\&}, 50
\texttt{\&}, 53
\geq, 53
\%, 8
&$, 50
\&$, 53
accessor, 86
ackerman, 21
actual parameters, 69
actual TCC, 69
\texttt{AND}, 50
application expressions, 54
assuming TCC, 71
assumptions, 24
auto-rewrites, 34–36
axioms, 28–34
type constructor, 31
coprime types, see cotuple type
\texttt{COROLLARY}, 25
cotuple expression, 58
cotuple types, 41, 45
coverage property, 61
coverage TCC, 61
curried applications, 54
datatype
accessor, 86
constructor, 86
constructor specification, 86
recognizer, 86
declaration, 11–36
binding, 12
body, 12
constants, 17
formulas, 24–25
identifier, 12
kind, 11
\texttt{expr}, 11
\texttt{prop}, 11
\texttt{theory}, 11
type, 11
library, 34
local, 11
multiple, 12
top-level, 11
variables, 17
dependent types, 41, 46, 55
disequality, 50
disjointness property, 61
disjointness TCC, 61
domain mismatch TCC, 44
empty type, 15, 42, 43
enumeration types, 12, 14–15
equality, 50
existence TCC, 15
\texttt{EXISTS}, 54
exporting, 11
\texttt{EXPORTING}, 69–70
expression, 65
context, 43
conversion
\texttt{extend\_bool}, 30
\texttt{lambda}, 30, 31
\texttt{restrict}, 29
coversion, 28–34
type constructor, 31
coprime types, see cotuple type
\texttt{COROLLARY}, 25
cotuple expression, 58
cotuple types, 41, 45
coverage property, 61
coverage TCC, 61
curried applications, 54
datatype
accessor, 86
constructor, 86
constructor specification, 86
recognizer, 86
declaration, 11–36
binding, 12
body, 12
constants, 17
formulas, 24–25
identifier, 12
kind, 11
\texttt{expr}, 11
\texttt{prop}, 11
\texttt{theory}, 11
type, 11
library, 34
local, 11
multiple, 12
top-level, 11
variables, 17
dependent types, 41, 46, 55
disequality, 50
disjointness property, 61
disjointness TCC, 61
domain mismatch TCC, 44
empty type, 15, 42, 43
enumeration types, 12, 14–15
equality, 50
existence TCC, 15
\texttt{EXISTS}, 54
exporting, 11
\texttt{EXPORTING}, 69–70
expression, 65
expressions, 49
extend_bool conversion, 30
f91, 21
FACT, 25
factorial, 19
FALSE, 50
fixed inductive variable, 24
FORMULA, 25
formula declarations, 24–25
free variables, 25
FROM, 13
function
  partial, 3
  total, 3
function types, 41, 43–44
generic reference, 71
higher-order logic, 3
identifiers, 7
IF-THEN-ELSE, 52
IFF, 50
implies, 50
IMPORTING, 34
importing, 11
importings, 70–71
inductive definition, 22–24
interpreted type, 12
interpreted type declarations, 14
judgements, 25–28
LAMBDA, 54
lambda conversion, 30, 31
lambda expressions, 55
LAW, 25
LEMMA, 25
LET expressions, 55
LIBRARY, 34
library declaration, 34
macros, 21–22
mapping, 75
measure, 19
measure function, 18
monotonicity TCC, 23
mutual exclusion property, 61
mutual recursion, 18
name equivalence, 14, 41
nonempty type, 15
NONEMPTY_TYPE, 15, 16
NOT, 50
number-name-judgement, 26
numerals, 53
  overloading, 53
numeric expressions, 53
obligations, 24
operator symbols, 49
OR, 50
ordinal, 19
overloading, 11
overloading numerals, 53
override expression, 59
parameterized type names, 14
polymorphism, 69
positive occurrence, 23
postulate, 25
precedence, 49
pred, 44
predicate subtype, 4
projection expressions, 56
PROPOSITION, 25
PVS Context, 34
quantified expressions, 55
real, 53
recognizer, 86
record accessors, 46
record expressions, 57
record types, 41, 46
recursion
  mutual, 18
recursive definitions, 18–21
recursive signature, 19
reserved words, 7
restrict conversion, 29
set theory, 3
setof, 44
special symbols, 8
stacks example, 4
string expressions, 53
structural equivalence, 14, 41
SUBLEMMA, 25
subtype judgement, 26, 27
subtype predicate, 41, 42
subtype TCC, 43
subtypes, 13, 41–43
sum types, see cotuple type
super type, see cotuple type
syntax
  conventions, 93
  declarations, 11
table consistency, 61
tables, 60–65
TCC, 43
  actual, 69
  assuming, 71
  cases, 58, 91
  coverage, 61
  disjointness, 61
  domain mismatch, 44
  existence, 15
  monotonicity, 23
  subtype, 43
termination, 19
termination-subtype, 19
  well-founded, 19
termination TCC, 19
termination-subtype TCC, 19
THEOREM, 25
theorems, 24
theories, 67
theory abbreviations, 71
theory instance, 71
theory interpretations, 75
theory parameters, 69
total function, 18
TRUE, 50
tuple expressions, 56
tuple types, 41, 44
type, 41–47
application, 41
cotuple, 41, 45
dependent, 41, 46
empty, 15–17, 42, 43
enumeration, 12, 14–15
extension, 47
function, 41, 43–44
interpreted, 12, 14
name, 12
nonempty, 15–17
ordinal, 19
record, 41, 46
subtype, 13, 41–43
supertype, 13
tuple, 41, 44
uninterpreted, 12
uninterpreted subtype, 12
NONEMPTY_TYPE, 12
TYPE, 12
type application, 41
type constructor conversion, 31
type constructors, 41
type declarations, 12–15
type expressions, 41
TYPE+, 12, 15, 16
uninterpreted subtype, 12–13
uninterpreted type, 12, 12
universal closure, 25
update expression, 59
variables, 17
well-founded order relation, 18
well-founded TCC, 19
WHEN, 50
WHERE expressions, 55
with expression, 59