# PVS Language Reference

Version 2.4 • November 2001

S. Owre
N. Shankar
J. M. Rushby
D. W. J. Stringer-Calvert
{Owre,Shankar,Rushby,Dave_SC}@csl.sri.com
http://pvs.csl.sri.com/

SRI International
Computer Science Laboratory • 333 Ravenswood Avenue • Menlo Park CA 94025

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

# Chapter 1

# Introduction

PVS is a *P*rototype *V*erification *S*ystem for the development and analysis of formal specifications. The PVS system consists of a specification language, a parser, a typechecker, a prover, specification libraries, and various browsing tools. This document primarily describes the specification language and is meant to be used as a reference manual. The *PVS System Guide* [9] is to be consulted for information on how to use the system to develop specifications and proofs. The *PVS Prover Guide* [13] is a reference manual for the commands used to construct proofs.

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

The following sections provide more details on the various features of the language. The lexical aspects of the language are detailed in Section 2. Section 3 describes declarations, Section 4 describes type expressions, Section 5 describes expressions, and Section 6 describes theories, theory parameters, and imports and exports of names. Section 7 describes names and name resolution, and Section 8 describes the datatype facility of PVS. Finally, Appendix A provides the grammar of the language.

## 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 the function, record, and tuple type constructions. The *terms* of the language can be constructed using function application, lambda abstraction, and record and tuple construction.

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 introduced. 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:real | x /= 0}`. One benefit of such subtyping is that when an operation is not defined on all the elements 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 discharge 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 facility 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 explicated 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 specification 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. However, 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, sequences, 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[1] 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 [8]). 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

[1] The PVS semantics are presented in a technical report [12].

*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 [7], LUTINS [4], RAISE [5], Beeson [1] and Scott [11], 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.

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 8.1, page 72.

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 *nonempty* 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

```
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`

`nonemptystack?` is then declared on elements of type `stack`; it is `true` for a given `stack` element iff[2] that element is not equal to `empty`. 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 as a type 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, y: t, x: t):
```

[2] Iff is short for "if and only if".

```
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, the theory

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

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

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.

# 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. They may be arbitrarily long, constrained only by the limits imposed by the underlying Common Lisp system. Identifiers are case-sensitive; `FOO`, `Foo`, and `foo` are different identifiers. PVS strings contain any ASCII character: to include a `"` in the string, use `\"` and to include a `\` use `\\`.

| | | |
|---|---|---|
| *Ids* | ::= | *Id*++ ',' |
| *Id* | ::= | *Letter* *IdChar*+ |
| *Number* | ::= | *Digit*+ |
| *String* | ::= | `"` *ASCII-character** `"` |
| *IdChar* | ::= | *Letter* \| *Digit* \| `_` \| `?` |
| *Letter* | ::= | `A` \| ... \| `Z` \| `a` \| ... \| `z` |
| *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.

| | | | | |
|---|---|---|---|---|
| AND | CONJECTURE | FACT | LET | TABLE |
| ANDTHEN | CONTAINING | FALSE | LIBRARY | THEN |
| ARRAY | CONVERSION | FORALL | MACRO | THEOREM |
| ASSUMING | CONVERSION+ | FORMULA | MEASURE | THEORY |
| ASSUMPTION | CONVERSION- | FROM | NONEMPTY_TYPE | TRUE |
| AUTO_REWRITE | COROLLARY | FUNCTION | NOT | TYPE |
| AUTO_REWRITE+ | DATATYPE | HAS_TYPE | O | TYPE+ |
| AUTO_REWRITE- | ELSE | IF | OBLIGATION | VAR |
| AXIOM | ELSIF | IFF | OF | WHEN |
| BEGIN | END | IMPLIES | OR | WHERE |
| BUT | ENDASSUMING | IMPORTING | ORELSE | WITH |
| BY | ENDCASES | IN | POSTULATE | XOR |
| CASES | ENDCOND | INDUCTIVE | PROPOSITION | |
| CHALLENGE | ENDIF | JUDGEMENT | RECURSIVE | |
| CLAIM | ENDTABLE | LAMBDA | SUBLEMMA | |
| CLOSURE | EXISTS | LAW | SUBTYPES | |
| COND | EXPORTING | LEMMA | SUBTYPE_OF | |

Figure 2.2: PVS Reserved Words

<table>
<tr><td>#</td><td>*</td><td>:)</td><td>=></td><td>\/</td><td>|=</td></tr>
<tr><td>##</td><td>**</td><td>::</td><td>></td><td>]</td><td>|></td></tr>
<tr><td>#)</td><td>+</td><td>:=</td><td>>=</td><td>]|</td><td>|[</td></tr>
<tr><td>#]</td><td>++</td><td>;</td><td>>></td><td>^</td><td>|]</td></tr>
<tr><td>%</td><td>,</td><td><</td><td>>>=</td><td>^^</td><td>||</td></tr>
<tr><td>&</td><td>-</td><td><<</td><td>@</td><td>`</td><td>|}</td></tr>
<tr><td>&&</td><td>-></td><td><<=</td><td>@@</td><td>{</td><td>}</td></tr>
<tr><td>(</td><td>.</td><td><=</td><td>[</td><td>{|</td><td>~</td></tr>
<tr><td>(#</td><td>/</td><td><=></td><td>[#</td><td>{||}</td><td></td></tr>
<tr><td>(:</td><td>//</td><td><></td><td>[]</td><td>|</td><td></td></tr>
<tr><td>(|</td><td>/=</td><td><|</td><td>[|</td><td>|)</td><td></td></tr>
<tr><td>(||)</td><td>/\</td><td>=</td><td>[||]</td><td>|-</td><td></td></tr>
<tr><td>)</td><td>:</td><td>==</td><td>\</td><td>|-></td><td></td></tr>
</table>

Figure 2.3: PVS Special Symbols

The whitespace characters are space, tab, newline, return, and newpage; 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 given declarations. Some of them have declarations given in the prelude. Any of

<table>
<tr><td>##</td><td>//</td><td><|</td><td>AND</td><td>ORELSE</td><td>{||}</td></tr>
<tr><td>&</td><td>/=</td><td>=</td><td>ANDTHEN</td><td>TRUE</td><td>|-</td></tr>
<tr><td>(||)</td><td>/\</td><td>==</td><td>FALSE</td><td>WHEN</td><td>|=</td></tr>
<tr><td>*</td><td><</td><td>=></td><td>IF</td><td>XOR</td><td>|></td></tr>
<tr><td>**</td><td><<</td><td>></td><td>IFF</td><td>[]</td><td>~</td></tr>
<tr><td>+</td><td><<=</td><td>>=</td><td>IMPLIES</td><td>[||]</td><td></td></tr>
<tr><td>++</td><td><=</td><td>>></td><td>NOT</td><td>\/</td><td></td></tr>
<tr><td>-</td><td><=></td><td>>>=</td><td>O</td><td>^</td><td></td></tr>
<tr><td>/</td><td><></td><td>@@</td><td>OR</td><td>^^</td><td></td></tr>
</table>

Figure 2.4: PVS Definable Symbols

these may be (re)declared any number of times, though this may lead to ambiguities. 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.

The symbol pairs `[|` and `|]`, `(|` and `|)`, and `{|` and `|}` are available as outfix operators. They are declared using `[||]`, `(||)`, and `{||}`, respectively. For example, with the declaration `[||]: [bool, int -> int]` the outfix term `[| TRUE, 0 |]` is equivalent to the prefix form `[||](TRUE, 0)`.

# 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 and conversions. Each declaration has an *identifier* and belongs to a unique theory. Declarations also have a body which indicates the *kind* of the declaration and provides the signature and definition of the declaration. *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 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 only be referenced in proofs (see the PVS Prover Guide [13]). Variable, constant, and recursive definition declarations are of kind *expr*; these may be referenced in expressions and actual parameters. Newly introduced names need

[1]Thus mutual recursion is not directly supported. The effect can be achieved with a simple recursive function that has an argument that serves as a switch for selecting between two or more expressions.

only be unique within a kind, as there is no way, for example, to use an expression where a type is expected.[2]

Declarations 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 one body is specified for multiple identifiers; for example

```
x, y, z: VAR int
```

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

```
x: VAR int
y: VAR int
z: VAR int
```

In the rest of this section we describe declarations for types, variables, constants, recursive definitions, and formulas. The declarations for theory parameters and theory abbreviations are given in Section 6. Figure 3.1 gives the grammar 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: TYPE`
- *uninterpreted subtype declaration*: `S: TYPE FROM T`
- *interpreted type declaration*: `T: TYPE = int`
- *enumeration type declarations*: `T: 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 `TYPE`, `NONEMPTY_TYPE`, or `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,

```
T1, T2, T3: TYPE
```

[2]The only exception is actual parameters to theories, since theories may be instantiated with types or expressions.

```
LibDecl              ::= Ids : LIBRARY [=] String
TheoryAbbrDecl       ::= Ids : THEORY = TheoryName
TypeDecl             ::= Id [{, Ids} | Bindings] :
                           {TYPE | NONEMPTY_TYPE | TYPE+}
                           [ {= | FROM} TypeExpr [CONTAINING Expr] ]
VarDecl              ::= IdOps : VAR TypeExpr
ConstDecl            ::= IdOp [{, IdOps } | Bindings+] : TypeExpr [= Expr]
RecursiveDecl        ::= IdOp [{, IdOps } | Bindings+] : RECURSIVE
                         TypeExpr = Expr MEASURE Expr [BY Expr]
MacroDecl            ::= IdOp [{, IdOps } | Bindings+] : MACRO
                           TypeExpr = Expr
InductiveDecl        ::= IdOp [{, IdOps } | Bindings+] : INDUCTIVE
                           TypeExpr = Expr
Assumption           ::= Ids : ASSUMPTION Expr
FormulaDecl          ::= Ids : FormulaName Expr
Judgement            ::= SubtypeJudgement | ConstantJudgement
SubtypeJudgement     ::= [IdOp :] JUDGEMENT TypeExpr++',' SUBTYPE_OF TypeExpr
ConstantJudgement    ::= [IdOp :] JUDGEMENT ConstantReference++','
                         HAS_TYPE TypeExpr
ConstantReference    ::= Number | {Name Bindings*}
Conversion           ::= {CONVERSION | CONVERSION+ | CONVERSION-}
                         { Name [: TypeExpr] }++','
AutoRewriteDecl      ::= {AUTO_REWRITE | AUTO_REWRITE+ | AUTO_REWRITE-}
                         RewriteName++','
RewriteName          ::= Name [! [!]] [: {TypeExpr | FormulaName} ]
Bindings             ::= ( Binding++',' )
Binding              ::= TypedId | { ( TypedIds ) }
TypedIds             ::= IdOps [: TypeExpr] [| Expr]
TypedId              ::= IdOp [: TypeExpr] [| Expr]
```

Figure 3.1: Declarations Syntax

introduces three new pairwise disjoint types. If desired, further constraints may be put on these types by means of `AXIOM`s (see Section 3.7).

It should be emphasized that uninterpreted types are important in providing the right level of abstraction in a specification. Specifying the type in detail 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: TYPE FROM t
```

This introduces an uninterpreted *subtype* `s` of the *supertype* `t`. This has the same meaning as

```
s_pred: [t -> bool]
s: TYPE = (s_pred)
```

in which a new predicate is introduced in the first line and the type `s` is declared as a *predicated* 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 also.

### 3.1.3 Interpreted Type Declarations

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

```
intfun: TYPE = [int -> int]
```

introduces the type name `intfun` as an abbreviation for the type of functions with integer domain and range. Because PVS uses *structural equivalence* instead of *name equivalence*, any type expression `T` involving `intfun` is equivalent to the type expression obtained by substituting `[int -> int]` for `intfun` in `T`. The available type expressions include subtypes, function types, tuple types, and record types. These are described in Section 4 on page 37.

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

```
subrange(m, n: int): TYPE = {i:int | m <= i AND i <= n}
```

and `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 `subrange` may be used to declare a different type without any ambiguity, as long as the number of parameters is different.

### 3.1.4 Enumeration Type Declarations

Enumeration type declarations are of the form

[3]This is described in Section 4.1 (page 37).

```
enum: TYPE = {e_1,..., 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?
       ⋮
  e_n: e_n?
END enum
```

explained in Section 8. 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.

### 3.1.5 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 subtypes, 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,[4] 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.

[4] If the type T is empty, then the following two equivalences hold:
(FORALL (x: T): p(x)) IFF TRUE, (EXISTS (x: T): p(x)) IFF FALSE

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

- 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 subtype 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 (e.g. `[t: TYPE, f: [t -> 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.

### 3.1.6 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 type-checker 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:

```
t1: TYPE = {x: int | x > 2}
t2: TYPE = {x: int | x > 2}
c1: t1
```

only marks the first type (`t1`). 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 `FORALL` and `LAMBDA`. Such local declarations "shadow" any earlier declarations. For example, in

```
x: VAR bool
f: FORMULA (FORALL (x: int): (EXISTS (x: nat): p(x)) AND q(x))
```

The occurrence of `x` as an argument to `p` is of type `nat`, shadowing the one of type `int`. Similarly, the occurrence of `x` as an argument to `q` is of type `int`, shadowing the one of type `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 *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 4.1, page 37). Here are some examples of constant declarations:

```
n: int
c: int = 3
f: [int -> int] = (lambda (x: int): x + 1)
g(x: int): 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 `AXIOM`s. The other three constants are interpreted. Each is equivalent to specifying two declarations: *e.g.*, the third line is equivalent to

```
f: [int -> int]
f: AXIOM  f = (LAMBDA (x: int): x + 1)
```

except that the definition is guaranteed to form a *conservative extension* of the theory. Thus the theory remains consistent after the declaration is given if it was consistent before.

The declarations for `f` and `g` above are two different ways to declare the same function. This extends to more complex arguments, for example

```
f: [int -> [int, nat -> [int -> int]]] =
    (LAMBDA (x: int): (LAMBDA (y: int), (z: nat): (LAMBDA (w: int):
      x * (y + w) - z)))
```

is equivalent to

```
f(x: int)(y: int, z: nat)(w: int): int = x * (y + w) - z
```

This can be shortened even further if the variables are declared first:

```
x, y, w: VAR int
z: VAR nat
f(x)(y,z)(w): int = x * (y + w) - z
```

Finally, a mix of predeclared and locally declared variables is possible:

```
x, y: VAR int
f(x)(y,(z: nat))(w: int): int = x * (y + w) - z
```

Note the parentheses around `z: nat`; without these, `y` would also be treated as if it were declared to be of type `nat`.

A construct that is frequently encountered when subtypes are involved is shown by this example

```
f(x: {x: int | p(x)}): int = x + 1
```

There are two useful abbreviations for this expression. In the first, we use the fact that the type `{x: int | p(x)}` is equivalent to the type expression `(p)` when `p` has type `[int -> bool]`, and we can write

```
f(x: (p)): int = x + 1
```

The second form of abbreviation basically removes the set braces and the redundant references to the variable, though extra parentheses are required:

```
f((x: int | p(x))): int = x + 1
```

Which of these forms to use is mostly a matter of taste; in general, choose the form that is clearest to read for a given declaration.

Note that functions with range type `bool` are generally referred to as *predicates*, and can also be regarded as relations or sets. For example, the set of positive odd numbers can be characterized by a predicate as follows:

```
odd: [nat -> bool] = (LAMBDA (n: nat): EXISTS (m: nat): n = 2 * m + 1)
```

PVS allows an alternate syntax for predicates that encourages a set-theoretic interpretation:

```
odd: [nat -> bool] = {n: nat | EXISTS (m: nat): n = 2 * m + 1}
```

## 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:

```
factorial(x: nat): RECURSIVE nat =
  IF x = 0 THEN 1 ELSE x * factorial(x - 1) ENDIF
  MEASURE (LAMBDA (x: nat): x)
```

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

```
factorial_TCC2: OBLIGATION
  FORALL (x: nat): 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:

```
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.

```
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

```
[[(var?), term] -> [{z1: term | z1 << s} -> term]]
```

Hence the signature of `subst(x, y)` is `[{z1: term | z1 << s} -> term]`, and map is instantiated to `map[{z1: term | z1 << s}, term]`, which leads to the TCC

```
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[term, term]`, as this leads to a TCC that says every `term` is `<< 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,

[5]Some of these may be subsumed by earlier TCCs, and hence will not be displayed with the `M-x show-tccs` command.

```
  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: term|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`.

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

The type of the measure function is `[real -> [nat, nat -> 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

```
 [real -> [{z: [nat, nat] | z`1 < x} -> [boolean -> 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 sounde 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.

```
 fbad: RECURSIVE [nat -> nat] = fbad
  MEASURE lambda (n: nat): n
```

This leads an unprovable TCC.

```
 fbad_TCC1: OBLIGATION FORALL (x1: nat, x: nat): x < x1;
```

The TCC results from the comaprison of the expected type `[nat -> nat]` to the derived type `[{z: nat | z < x1} -> 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, in Figure 3.2 the termination TCC is

```
f91_TCC5: OBLIGATION
  FORALL (i: nat,
          v: [i1:
               {z: nat |
                        (IF z > 101 THEN 0 ELSE 101 - z ENDIF) <
```

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

  END f91
```

Figure 3.2: Theory `f91`

```
                         (IF i > 101 THEN 0 ELSE 101 - i ENDIF)} ->
              {j: nat | IF i1 > 100 THEN j = i1 - 10 ELSE j = 91 ENDIF}]):
    NOT i > 100 IMPLIES
     IF i > 100 THEN v(v(i + 11)) = i - 10 ELSE v(v(i + 11)) = 91 ENDIF;
```

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 4.5.

```
ackerman: THEORY
  BEGIN

  m, n: VAR nat
  ackmeas(m, n): ordinal =
      (IF m = 0 THEN zero
       ELSIF n = 0 THEN add(m, add(1, zero, zero), zero)
       ELSE add(m, add(1, zero, zero), add(n, zero, zero))
       ENDIF)

  ack(m, n): RECURSIVE nat =
      (IF m = 0 THEN n + 1
       ELSIF n = 0 THEN ack(m - 1, 1)
       ELSE ack(m - 1, ack(m, n - 1))
       ENDIF)
    MEASURE ackmeas

  END ackerman
```

Figure 3.3: Theory `ackerman`

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 3.3,[6] there are two termination TCCs generated (along with a number of subtype TCCs). The first termination TCC is

```
ack_TCC2:
  OBLIGATION
    (FORALL m, n:
      NOT m = 0 AND n = 0 IMPLIES ackmeas(m - 1, 1) < ackmeas(m, n))
```

and corresponds to the first recursive call of `ack` in the body of `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 `ack` is less than the measure applied to the initial arguments of `ack`. Note that the `<` in this expression is over the `ordinals`, not the `reals`.

## 3.5 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 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.[7] For example, after the declaration

```
N: MACRO nat = 100
```

any reference to `N` is now automatically replaced by `100`, including such forms as `below[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.

## 3.6 Inductive Definitions

PVS provides support for constructing inductive definitions. Inductive definitions are usually presented by giving some rules for generating elements of a set and then stating that an object is in the set only if it has been generated according to the rules; hence, it is the smallest set closed under the rules. Inductive definitions are similar to recursive definitions, in that both involve induction, and both must satisfy additional constraints to guarantee that they are total.

[6] There are ways of specifying `ackerman` using higher-order functionals, in which case the measure is again on the natural numbers.

[7] This is similar to the `==` form of EHDM.

The even integers provide a simple example of an inductive definition:[8]

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

With this definition, it is easy to prove, for example, that `0` or `1000` are even, simply by expanding the definition enough times.[9] 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:

```
even_weak_induction: AXIOM
      (FORALL (P: [nat -> boolean]):
         (FORALL (n: nat): n = 0 OR (n > 1 AND P(n - 2)) IMPLIES P(n))
             IMPLIES (FORALL (n: nat): even(n) IMPLIES P(n)));

even_induction: AXIOM
      (FORALL (P: [nat -> boolean]):
         (FORALL (n: nat):
            n = 0 OR (n > 1 AND even(n - 2) AND P(n - 2)) IMPLIES P(n))
             IMPLIES (FORALL (n: nat): 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 [13].

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

```
f1(n,m:int) INDUCTIVE int = n
f2(n,m:int)(x,y:int)(z:int): INDUCTIVE [int,int,int -> bool] =
    LAMBDA (a,b,c:int): n = m IMPLIES f2(n,m)(x,y)(z)(a,b,c)
```

`f1` is illegal, while `f2` returns a boolean value if applied to enough arguments, hence is valid.

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

```
f: [nat, bool -> bool]
```

[8] This is an alternative to the more traditional definition of `even?` in the prelude.

[9] In the latter case, `(apply (repeat (expand "even")))` is a good strategy to use, though it should be used with care since it does not terminate in all cases.

```
G(n:nat): INDUCTIVE bool =
  n = 0 OR f(n, G(n-1))
```

the monotonicity TCC has the form

```
(FORALL (P1: [nat -> boolean], P2: [nat -> boolean]):
   (FORALL (x: nat): P1(x) IMPLIES P2(x))
       IMPLIES
     (FORALL (x: nat):
        x = 0 OR f(x, P1(x - 1)) IMPLIES x = 0 OR f(x, P2(x - 1))));
```

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) : INDUCTIVE bool =
   R(x, y) OR (EXISTS z: TC(R)(x, z) AND 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

```
FORALL (F, P):
 (FORALL (N):
   inductive_body(N)[P/def] IMPLIES P(N))
    IMPLIES
   (FORALL (N): def(V) IMPLIES P(N))
```

In the case of `TC`, this becomes

```
TC_weak_induction: AXIOM
      (FORALL (R: relation, P: [[T, T] -> boolean]):
         (FORALL (x: T, y: T):
            R(x, y) OR (EXISTS z: (P(x, z) AND P(z, y))) IMPLIES P(x, y))
             IMPLIES (FORALL (x: T, y: T): TC(R)(x, y) IMPLIES P(x, y)));
```

## 3.7 Formula Declarations

Formula declarations introduce *axioms*, *assumptions*, *theorems*, and *obligations*. The identifier associated with the declaration may be referenced during proofs (see the `lemma` command in the PVS Prover Guide [13]). The expression that makes up the body of the formula is a boolean expression. Axioms, assumptions, and obligations are introduced with the keywords `AXIOM`, `ASSUMPTION`, and `OBLIGATION`, respectively. Axioms may also be introduced using the keyword `POSTULATE`. In the prelude postulates are used to indicate axioms that are provable by the decision procedures, but not from other axioms. Theorems may be introduced with any of the keywords `CHALLENGE`,

CLAIM, CONJECTURE, COROLLARY, FACT, FORMULA, LAW, LEMMA, PROPOSITION, SUBLEMMA, or THEOREM.

Assumptions are only allowed in assuming clauses (see Section 6.5). Obligations are generated by the system for TCCs, and cannot be specified by the user. Axioms are treated specially when a proof is analyzed, in that they are not expected to have an associated proof. Otherwise they are treated exactly like theorems. All the keywords associated with theorems have the same semantics, they are there simply to allow for greater diversity in classifying formulas.

Formula declarations may contain free variables, in which case they are equivalent to the universal closure of the formula.[10] In fact, the prover actually uses the universal closure when it introduces a formula to a proof. Formula declarations are the only declarations in which free variables are allowed.

## 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*[11] provide a means for controlling this by allowing properties of operators on subtypes to be made available to the typechecker. There are two kinds of judgements available in PVS. 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 Constant 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

```
JUDGEMENT c, 17 HAS_TYPE (prime?)
```

simply states that the constant c and the number 17 are both prime numbers. This declaration leads to the TCC formulas prime?(c) and prime?(17), but in any context in which this declaration is visible, these TCCs will not be generated. Thus no TCCs are generated for the formula F in

```
RP: [(prime?), (prime?) -> bool]
F: FORMULA RP(c, 17) IMPLIES 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 HAS_TYPE keyword. Here is an example that illustrates the need for this kind of judgement:

[10] The universal closure of a formula is obtained by surrounding the formula with a FORALL binding operator whose bindings are the free variables of the formula. For example, the universal closure of p(x,y) => q(z) is (FORALL x,y,z: p(x,y) => q(z)) (assuming x, y and z resolve to variables).

[11] We prefer this spelling, though many spell checkers do not.

[12] Remember that all numbers are implicitly declared to be of type real.

```
x, y: VAR real
f(x,y): real = x*x - y*y
n: int = IF f(1,2) > 0 THEN f(4,3) ELSE f(3,2) ENDIF
```

This leads to two TCCs:

```
n_TCC1: OBLIGATION
  f(1, 2) > 0 IMPLIES
    rational_pred(f(4, 3)) AND integer_pred(f(4, 3))
n_TCC2: OBLIGATION
  NOT f(1, 2) > 0 IMPLIES
    rational_pred(f(3, 2)) 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 *ad hoc* solution of adding new overloaded declarations for `f`:

```
i, j: VAR nat
f(i, j): 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_int_is_int: JUDGEMENT f(i, j: int) HAS_TYPE int
```

This generates the TCC

```
f_int_is_int: FORALL (x:int, y:int):
                rational_pred(f(x, y)) 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_int: JUDGEMENT f HAS_TYPE [int, int -> int]
```

In this case, the TCC generated is unprovable:

```
f_int: OBLIGATION
  ((FORALL (x: real): rational_pred(x) AND integer_pred(x)) AND
    (FORALL (x: real): rational_pred(x) AND integer_pred(x)))
   AND
   (FORALL (x1: [real, real]): rational_pred(f(x1)) AND integer_pred(f(x1)));
```

A warning is generated when simple constant judgements are declared to be of a function type.[14] In addition, this judgement will not help with the declaration `n` above; it can only be used in higher-order functions, for example:

```
F: [[int, int -> int] -> bool]
FF: 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: real)(z: real): real
f_closed: JUDGEMENT f(x, y: nat)(z: int) HAS_TYPE int
```

[13]This is one of the motivations for providing the `M-x show-expanded-sequent` command.

[14]Earlier versions of PVS simply interpreted this form as a closure condition, but this is less flexible.

```
f2_closed: JUDGEMENT f(x, y: int) 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: VAR real
x_times_x_is_nonneg: JUDGEMENT *(x, x) HAS_TYPE nonneg_real
c: real
x_times_c_is_even: JUDGEMENT *(x, c) 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:

```
nonzero_real: NONEMPTY_TYPE = {r: real | r /= 0} CONTAINING 1
rational: NONEMPTY_TYPE FROM real
nonneg_rat: NONEMPTY_TYPE = {r: rational | r >= 0} CONTAINING 0
posrat: NONEMPTY_TYPE = {r: nonneg_rat | r > 0}  CONTAINING 1
/: [real, nonzero_real -> real]
```

For `r` of type `real` and `q` of type `posrat`, the expression `r / q` leads to the TCC `q /= 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`:

```
JUDGEMENT posrat SUBTYPE_OF nzrat
JUDGEMENT nzrat SUBTYPE_OF nzreal
```

With these judgements, TCCs will not be generated for any denominator that is of type `posrat`. With the (prelude) judgement declarations

```
nnrat_plus_posrat_is_posrat:   JUDGEMENT +(nnx, py) HAS_TYPE posrat
posrat_times_posrat_is_posrat: JUDGEMENT *(px, py)  HAS_TYPE posrat
```

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

Given a subtype judgement declaration of the form

```
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 | p(x)`}, where `ST` is the least compatible type of `S` and `T`, and a TCC will is generated of the form `FORALL (x:S): p(x)`. Remember that subtyping on functions only works on range types, so the subtype judgement

```
  JUDGEMENT [nat -> nat] SUBTYPE_OF [int -> int]
```

leads to the unprovable TCC

```
FORALL (x1:nat, y1:int): y1 >= 0 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*.

### 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[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. 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: 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(restrict[int,nat,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).

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 `extend(s)` is the type-extended version of the original set `s`, we have `extend(s)(x) = s(x)` if `x` is in the subtype `S`, and `extend(s)(x) = 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.

```
extend [T: TYPE, S: TYPE FROM T, R: TYPE, d: R]: THEORY
 BEGIN
  f: VAR [S -> R]
  t: VAR T
```

[15] The `M-x prettyprint-expanded` command.

```
    extend(f)(t): R = IF S_pred(t) THEN f(t) ELSE d ENDIF
   END extend
```

The function extend(f) has type [T -> R] and is constructed from the function f of type [S -> R] (where S is a subtype of T) by supplying the default value d whenever its argument is not in S (S_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.

```
extend_bool [T: TYPE, S: TYPE FROM T]: THEORY
 BEGIN
  CONVERSION extend[T, S, bool, false]
 END extend_bool
```

In the presence of this conversion, the type-incorrect formula B_app in the following specification

```
  b: [nat -> bool]
  B: [[int -> bool] -> bool]
  B_app: FORMULA B(b)
```

is automatically transformed to B(extend[int,nat,bool,false](b)).

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 state is an uninterpreted (nonempty) type, and a state variable v of type real is represented as a constant of type [state -> real].

```
  th: THEORY
   BEGIN
    state: TYPE+
    l: [state -> list[int]]
    x: [state -> real]
    b: [state -> bool]
    bv: VAR [state -> bool]
    s: VAR state
    box(bv): bool = FORALL s: bv(s)
    F1: FORMULA box(x > 1)
    F2: FORMULA box(b IMPLIES length(l) + 3 > x)
   END th
```

In this example, the formulas F1 and F2 are not type correct as they stand, but with a *lambda conversion*, triggered by the K_conversion in the PVS prelude, these formulas are converted to the forms

```
  F1: FORMULA box(LAMBDA (x1: state): x(x1) > 1)

  F2: FORMULA
    box(LAMBDA (x3: state):
          b(x3) IMPLIES
           (LAMBDA (x2: state):
```

```
(LAMBDA (x1: state): (LAMBDA (x: state): length(l(x)))(x1) + 3)
    (x2)
 > x(x2))
 (x3))
```

### 3.9.2 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)`. `C` is compatible if its type is `[D -> R]`, where `D` is compatible with $T_1$ and `R` is compatible with $T_2$.

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`, `c2` is the most recent conversion, followed by the conversions in theory `th2`, those in `th1`, and finally `c1`. Note that the relative order of the constant declarations (e.g., `c1` and `c2` above) doesn't matter, only the `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 have a composition looked for, define a new constant explicitly and include a conversion declaration for it. Here is an example:

```
T1, T2, T3: TYPE+
f1: [T1 -> T2]
f2: [T2 -> T3]
x: T1
g: [T3 -> bool]
CONVERSION f1, f2
F1: FORMULA g(x)
f3: [T1 -> T3] = f2 o f1
```

```
CONVERSION f3
F2: FORMULA g(x)
```

In this example, `F1` leads to a type error, but when we make the composition function `f3` a conversion, the same expression in `F2` applies the conversion rather than give a type error.

### 3.9.3 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 `mucalculus` 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 `ctlops`, 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` has the `CONVERSION+ c` declaration but no `CONVERSION-` declaration, 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. 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*[9]) is to use the `M-x load-prelude-library`, which augments the PVS prelude with the 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-`.

When a proof is initiated for a given formula, all the `AUTO_REWRITE+` names in the current context that haven't subsequently been removed by `AUTO_REWRITE-` declarations are collected and added to the initial proof state. The `AUTO_REWRITE-` declaration, in addition to removing `AUTO_REWRITE+` names, also affects the `auto-rewrite-theory`, `auto-rewrite-theories`, `auto-rewrite-theory-with-importings`, `simplify-with-rewrites`, `auto-rewrite-defs`, `install-rewrites`, `auto-rewrite-explicit`, `grind`, `induct-and-simplify`, `measure-induct-and-simplify`, and `model-check` commands, described in the Prover manual. These commands collect all definitions and formulas except those that appear in `AUTO_REWRITE-` delcarations. Thus suppose a theory `T` contains the lemmas `lem1`, `lem2`, and `lem3` and the declarations

```
AUTO_REWRITE+ lem1
AUTO_REWRITE- lem3
```

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

The syntax for auto-rewrite declarations is as follows.

| | | |
|---|---|---|
| *AutoRewriteDecl* | ::= | *AutoRewriteKeyword* *RewriteName*$_{++}$ ',' |
| *AutoRewriteKeyword* | ::= | `AUTO_REWRITE` \| `AUTO_REWRITE+` \| `AUTO_REWRITE-` |
| *RewriteName* | ::= | *Name* [ `!` [ `!` ] ] [ `:` { *TypeExpr* \| *FormulaName* } ] |

The `autorewrites` theory shows a simple example.

```
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, `foo!` or `foo!!`. Without the suffix, the rewrite is *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 *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 *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 `auto-rewrite`, `auto-rewrite!`, and `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.

# 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*, 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 8.

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 52 and 47, respectively), and as actual parameters in names (page 67). 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: TYPE = {x: s | p(x)}
```

| | | |
|---|---|---|
| *TypeExpr* | ::= | *Name* |
| | \| | *EnumerationType* |
| | \| | *Subtype* |
| | \| | *TypeApplication* |
| | \| | *FunctionType* |
| | \| | *TupleType* |
| | \| | *RecordType* |
| *EnumerationType* | ::= | { *IdOps* } |
| *Subtype* | ::= | { *SetBindings* `\|` *Expr* } |
| | \| | `(` *Expr* `)` |
| *TypeApplication* | ::= | *Name Arguments* |
| *FunctionType* | ::= | [`FUNCTION` \| `ARRAY`] |
| | | `[` { [*IdOp* `:`] *TypeExpr* }++ ',' `->` *TypeExpr* `]` |
| *TupleType* | ::= | `[` { [*IdOp* `:`] *TypeExpr* }++ ',' `]` |
| *RecordType* | ::= | `[#` *FieldDecls*++ ',' `#]` |
| *FieldDecls* | ::= | *Ids* `:` *TypeExpr* |

Figure 4.1: Type Expression Syntax

where `p` is a predicate on the type `s`.[1] This has the usual set-theoretical meaning, since types in PVS are modeled as sets. Subtypes may also be presented in an abbreviated form, by giving a predicate surrounded by parentheses:

```
t: 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

```
FORALL (i:int):
  (i >= 0 IMPLIES (EXISTS (j:int): j >= 0 AND j > i))
```

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

```
FORALL (i:nat): (EXISTS (j:nat): j > i))
```

where `nat` is defined in the prelude as

```
naturalnumber: NONEMPTY_TYPE = {i:integer | i >= 0} CONTAINING 0
nat: NONEMPTY_TYPE = naturalnumber
```

Subtype constructors consist of a *supertype* and an optional predicate on the supertype. The primary property of a subtype is that any element which belongs to

[1] If `x` has been previously declared as a variable of type `s`, then the "`: s`" may be omitted.

the subtype automatically belongs to the supertype. In addition, functions defined on a type automatically carry over to the subtype.

There are two *type-correctness conditions* (TCCs) associated with subtypes. The first concerns *empty types* as described in section 3.1.5. PVS allows empty types as long as only variables range over them. However, allowing declarations of constants involving empty types leads to inconsistencies. Whenever a constant is declared, the typechecker checks the types involved, and generates *existence* TCCs for those types which must be nonempty. For example,

```
f: [int -> {x:int | p(x)}]
```

leads to the TCC

```
f_TCC1: OBLIGATION (EXISTS (x: int): p(x))
```

These TCCs are recorded, so that the nonemptiness of a subtype need be established only once in a theory. However, the same TCC may be generated in different theories. In particular, if a theory declares a type but no constant of that type, then any theory which imports that theory and declares a constant of that type will generate the nonempty TCC.

A subtype may be guaranteed nonempty by providing a *witness*, in which case no existence TCC is generated, though typechecking the witness itself may generate a TCC. The witness is provided using the `CONTAINING` clause of a subtype expression, as illustrated in the following:

```
t: TYPE = {x: int | 0 < x AND x < 10} CONTAINING 1
```

In this case a TCC is generated with the witness in place of the existential variable, resulting in the trivial TCC[2]

```
t_TCC1: OBLIGATION 0 < 1 AND 1 < 10
```

The second TCC associated with subtypes is the *subtype* TCC,, which come about from the use of operations defined on subtypes which 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 `[real, nonzero_real -> real]`. So given the formula

```
div_form: FORMULA (FORALL (x,y: int):
                        x /= y IMPLIES (x - y)/(y - x) = -1)
```

the denominator is of type integer, but the signature for `/` demands a `nonzero_real`. The typechecker thus generates a *subtype* TCC whose conclusion is `(y - x) /= 0`. The premises of the TCC are obtained from the expression's *context*—the conditions which lead to the `/` operator—in this case `x /= y`. The TCC is then

```
div_form_TCC1: OBLIGATION
  (FORALL (x,y: int): x /= y IMPLIES (y - x) /= 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,

[2] This TCC will be proved automatically by PVS; see the `typecheck-prove` command in the PVS System Guide [9].

| Expression | Context for `E` |
|---|---|
| `IF A THEN E ELSE C ENDIF` | `A` |
| `IF A THEN B ELSE E ENDIF` | `NOT A` |
| `A AND E` | `A` |
| `A OR E` | `NOT A` |
| `A IMPLIES E` | `A` |

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 which ensure that the value of the expression does not depend on the value of functions applied outside their domain.

## 4.2 Function Types

Function types have three equivalent forms:

- `[`$\mathtt{t}_1$`, ..., `$\mathtt{t}_n$` -> t]`
- `FUNCTION[`$\mathtt{t}_1$`, ..., `$\mathtt{t}_n$` -> t]`
- `ARRAY[`$\mathtt{t}_1$`, ..., `$\mathtt{t}_n$` -> t]`

where each $\mathtt{t}_i$ is a type expression. An element of this type is simply a function whose domain is the sequence of types $\mathtt{t}_1$, ..., $\mathtt{t}_n$, 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 `[`$\mathtt{t}_1$`,...,`$\mathtt{t}_n$` -> t]` is a subtype of `[`$\mathtt{s}_1$`,...,`$\mathtt{s}_m$` -> s]` iff `s` is a subtype of `t`, $n = m$, and $\mathtt{s}_i = \mathtt{t}_i$ for $1 \leq i \leq n$. 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 [$t_1$, ..., $t_n$], where the $t_i$ 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 $i$th projection is of type [[$t_1$, ..., $t_n$] -> $t_i$]. 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 [$t_1$,..., $t_n$ -> t] and [[$t_1$,..., $t_n$] -> t] are equivalent; see Section 5.8 for more details.

## 4.4 Record Types

Record types are of the form [# $a_1$:$t_1$, ..., $a_n$:$t_n$ #]. 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.

## 4.5 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:

```
rem: [nat, d: {n: nat | n /= 0} -> {r: nat | r < d}]
pfn: [d:pred[dom], [(d) -> ran]]
stack: [# size: nat, elements: [{n:nat | n < size} -> t] #]
```

The declaration for `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 be associated only with those types on which later types depend.

The tuple type `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 `ran`. If the second component were given instead as a function of type `[dom -> ran]`, then equality no longer works as intended. For example, the absolute value function `abs` and the identity function `id` are the same on the domain `nat`, so we would like to have

```
((LAMBDA (x:int):x >= 0),abs) = ((LAMBDA (x:int):x >= 0),id)
```

but without the dependency this would be equivalent to `abs = id`.

`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 `pfn` example—in fact, if we were willing to use a tuple instead of a record encoding, `stack` could be declared as an instance of the type of `pfn`.

Another example, presented in [3] as a “challenge” to specification languages without partial functions, is easily handled with dependent types as shown below.

```
subp(i:int,(j:int | i >= j)): RECURSIVE int =
     (IF (i=j) THEN 0 ELSE (subp(i, j+1)+1) ENDIF)
  MEASURE i - j
```

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

```
subp_lemma: LEMMA subp(i, 0) = i OR subp(0, i) = i
```

This generates unprovable TCCs. In practice this is rarely a problem.

# 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 Figure 5.1 and Figure 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 precedences 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 of these operators may be overloaded, see Section 7 (page 67) 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,

```
AND: [state, state -> state] = (LAMBDA a,b: (LAMBDA t: a(t) AND b(t)));
OR: [state, state -> state] = (LAMBDA a,b: (LAMBDA t: a(t) OR b(t)));
```

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.

```
Expr ::= Number
       | String
       | Name
       | Id ! Number
       | Expr Arguments
       | Expr Binop Expr
       | Unaryop Expr
       | Expr ` { Id | Number }
       | ( Expr++',' )
       | (: 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
```

Figure 5.1: Expression syntax

## 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` $B \equiv B$ `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}
```

```
IfExpr          ::=  IF Expr THEN Expr
                     { ELSIF Expr THEN Expr } * ELSE Expr ENDIF
BindingExpr     ::=  BindingOp LambdaBindings : Expr
BindingOp       ::=  LAMBDA | FORALL | EXISTS | { IdOp ! }
LambdaBindings  ::=  LambdaBinding [ [,] LambdaBindings ]
LambdaBinding   ::=  IdOp | Bindings
SetBindings     ::=  SetBinding [ [,] SetBindings ]
SetBinding      ::=  { IdOp [ : TypeExpr ] } | Bindings
Assignment      ::=  AssignArgs { := | |-> } Expr
AssignArgs      ::=  Id [ ! Number ]
                 |   Number
                 |   AssignArg+
AssignArg       ::=  ( Expr++ ',' )
                 |   ‘ Id
                 |   ‘ Number
Selection       ::=  IdOp [ ( IdOps ) ] : Expr
TableExpr       ::=  TABLE [ Expr ] [ , Expr ]
                     [ ColHeading ]
                     TableEntry+ ENDTABLE
ColHeading      ::=  |[ Expr { | { Expr | ELSE } }+ ]|
TableEntry      ::=  { | [ Expr | ELSE ] }+ ||
LetBinding      ::=  { LetBind | ( LetBind++ ',' ) } = Expr
LetBind         ::=  IdOp Bindings* [ : TypeExpr ]
Arguments       ::=  ( Expr++ ',' )
```

Figure 5.2: Expression syntax (continued)

```
j: VAR {x: int | x > 100}
eq2: FORMULA i = j
```

`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

<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td>FORALL, EXISTS, LAMBDA, IN</td><td>None</td></tr>
<tr><td>|</td><td>Left</td></tr>
<tr><td>|-, |=</td><td>Right</td></tr>
<tr><td>IFF, <=></td><td>Right</td></tr>
<tr><td>IMPLIES, =>, WHEN</td><td>Right</td></tr>
<tr><td>OR, \/, XOR, ORELSE</td><td>Right</td></tr>
<tr><td>AND, &, &&, /\, ANDTHEN</td><td>Right</td></tr>
<tr><td>NOT, ~</td><td>None</td></tr>
<tr><td>=, /=, ==, <, <=, >, >=, <<, >>, <<=, >>=, <|, |></td><td>Left</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>Left</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>[], <></td><td>None</td></tr>
<tr><td>^, ^^</td><td>Left</td></tr>
<tr><td>`</td><td>Left</td></tr>
</table>

Figure 5.3: Precedence Table

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 prover provides commands which turn an equality into an `IFF`.

## 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

```
IF A THEN B
ELSIF C THEN D
```

```
  ELSE E
  ENDIF
```
translates to
```
  IF A THEN B
  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.

## 5.4 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 `[int -> [int, int -> int]]`, `f(0)(2,3)` yields an `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 TCC
```
  2 < 3
```
Application and tuple expressions have a special relation, due to the type equivalence of `[`$t_1$`,...,`$t_n$` -> t]` and `[[`$t_1$`,...,`$t_n$`] -> t]`, see Section 5.8 for details.

## 5.5 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: 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
```

In `ex1`, variables `x`, `y`, and `z` are all of type `real`. In `ex2` these same variables are of type `nat`, shadowing the global declarations. `ex3` illustrates the use of constraints; this is equivalent to the declaration

```
ex3: AXIOM FORALL (n: {n: 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

```
(LAMBDA (x: int): x + 3)
```

The type of this expression is the function type `[int -> int]`. In addition, when the range is `bool`, a lambda expression may be represented as a set expression; see Section 5.7.

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

```
FORALL (x: int), (y: {z: 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 $e_1 < e_2$.[2]

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

```
foo! (x : int) : e
```

is equivalent to

```
foo( LAMBDA (x : int) : e)
```

## 5.6 `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:

---

[1] Set expressions are also binding expressions; see Section 5.7 (page 49).

[2] Such 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.

```
LET x:int = 2, y:int = x * x IN x + y

x + y WHERE x:int = 2, y: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

```
(LAMBDA (x:int) : (LAMBDA (y: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

```
LET x = 2, y = x * x 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 used to determine the type of the variable. This is one way in which these expressions differ from their translation. It is usually better to either reference a variable or give the type, as the typechecker uses the "natural" type of the expression as the type of the variable, which can lead to extra TCCs.

The `LET` expression has a limited form of pattern matching over tuples. An example is

```
p: VAR [int, int]
+(p): int = LET (m, n) = p IN m + n
```

which is shorter than the equivalent

```
p: VAR [int, int]
+(p): int = LET m = p`1, n = p`2 IN m + n
```

## 5.7 Set Expressions

In PVS, sets of elements of a type `t` are represented as predicates, *i.e.*, functions from `t` to `bool`. The type of a set may be given as `[t -> bool]`, `pred[t]`, or `setof[t]`, which are all type equivalent.[3] The choice depends wholly on the intended use of the type. Similarly, a set may be given in the form `(LAMBDA (x: t): p(x))` or `{x: t | p(x)}`; these are equivalent expressions.[4] 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 `sets`.

[3]The prelude theory `defined_types` also defines `PRED`, `predicate`, `PREDICATE`, and `SETOF` as alternate equivalents.

[4]In fact, internally they are represented by the same abstract syntax, they simply print differently.

## 5.8 Tuple Expressions

A tuple expression of the type `[t`$_1$`,...,t`$_n$`]` has the form `(e`$_1$`,...,e`$_n$`)`. For example, `(2, TRUE, (LAMBDA x: x + 1))` is of type `[nat, bool, [nat -> 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,...,t_n] -> t]` $\equiv$ `[t_1,...,t_n -> 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 `injective?[[int,int,int],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 `[int,int,int]`.

## 5.9 Projection Expressions

The components of an expression whose type is a tuple can be accessed using the projection operators `‘1`, `‘2`, ... or `PROJ_1`, `PROJ_2`, .... 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,

```
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)
```

## 5.10 Record Expressions

Record expressions are of the form `(# a`$_1$` := e`$_1$`, ..., a`$_n$` := e`$_n$` #)`, which has type `[# a`$_1$`: t`$_1$`, ..., a`$_n$`: t`$_n$` #]`, where each `e`$_i$ is of type `t`$_i$. 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]
```

Record types may be dependent, and a record expression of a dependent type may lead to TCCs. 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. However, you can define your own functions to provide this capability, and even use the same name. For example:

```
point: TYPE = [# x, y: real #]
x(p:point): real = p`x
y(p:point): real = p`y
```

Now x and y may be provided wherever a function is expected. Note that this means that a subsequent expression of the form x(p) could be ambiguous, but the record field accessor is always preferred, so in practice such ambiguities don't arise.

## 5.11 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.

## 5.12 Override Expressions

Functions, tuples, and records may be "modified" by means of the override expression. The result of an override expression is a function, tuple, or record that is exactly the same as the original, except that at the specified arguments it takes the new values. For example,

```
identity 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

```
(id WITH [(0) := 1]) WITH [(1) := 2]  or
(LAMBDA x: IF x = 1 THEN 2 ELSIF 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

```
id 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; for example,

```
R: TYPE = [# a: int, b: [int -> [int, int]] #]
r1: R
r2: R = r1 WITH [`a := 0, `b(1)`2 := 4]
```

r2 is equivalent to

```
(# a := 0,
   b := LAMBDA (x: int):
          IF x = 1
          THEN (r1`b(x)`1, 4)
          ELSE r2`b(x)
          ENDIF #)
```

Another form of override expression is the maplet, indicated using |-> in place of :=. This is used to extend the domain of the corresponding element; for example, if `f:[nat -> int]` is given, then `f WITH [(-1) |-> 0]` is a function of type `[{i:int | i >= 0 OR i = -1} -> int]`. This is especially useful with dependent types, see Section 4.5. Domain extension is also possible for record and tuple types; for example, `r1 WITH [‘c |-> 3]` is of type `[# a : int, b : [int -> [int,int]], c: int #]`, and if `t1` is of type `[int, bool]`, then `t1 WITH [‘3 |-> 1]` is of type `[int, bool, int]`. It is an error to extend a tuple type such that gaps are left, so `t1 WITH [‘4 |-> 1]` is illegal, though `t1 WITH [‘3 |-> 1, ‘4 |-> 1]` is allowed.

In the past, the two forms of assignment (using `:=` and `|->`) were merely alternative notation, and domains would be extended automatically whenever the typechecker could not determine that the argument belonged to the domain. In most cases, extending the domain unnecessarily is harmless. However, when terms get large, the types can get cumbersome, slowing down the system dramatically. Even worse, when domains are extended and matched against a rewrite rule with the original type, the match can fail, and the automatic rewrite will not be triggered.

## 5.13 Coercion Expressions

Coercion expressions are of the form `expr :: type-expr`, indicating that the expression `expr` is expected to be of type `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

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

the coercion of `foo` to `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.

## 5.14 Tables

Many expressions are easier to express and to read when presented in tabular form, as described in [6, 10]. There are many types of tables, ten different interpretations are described in [10] 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[5] to be presented, and supports *table consistency conditions*. One of the consistency conditions

[5] In Parnas’ terms [10], these tables are *normal function tables* of one or two dimensions.

(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.

### 5.14.1 `COND` Expressions

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

```
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.

```
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.

```
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:

```
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 same `IF-THEN-ELSE` form shown above.

*cond-expr* ::= `COND` *Expr* `->` *Expr*$_+$ [`ELSE` `->` *Expr*]

Using `COND`, we can translate the following tabular specification of the *sign* function

| | $x < 0$ | $x = 0$ | $x > 0$ |
|---|---|---|---|
| $sign(x)$ | $-1$ | $0$ | $1$ |

into

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

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

| modes | conditions | |
|---|---|---|
| normal | false | true |
| low | not overridden | overridden |
| voter_failure | true | false |
| safety_injection | on | off |

can be represented as

```
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.

| state | input | |
|---|---|---|
| | x | y |
| a | a | b |
| b | b | b |

This translates to

```
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 `COND`s will have the form

```
 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

```
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.

```
LET
  x1 = COND input=x -> a, input=y -> b ENDCOND,
  x2 = COND input=x -> b, input=y -> b ENDCOND
IN
  COND state=a -> x1, state=b -> 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

```
CASES state OF
  a: CASES input OF x: a, y: b ENDCASES,
  b: CASES input OF x: b, y: b ENDCASES
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.

### 5.14.2 Table Expressions

The COND and CASES constructs (see datatypes on page 71) 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.

```
   TABLE
%                ---------------------
               |[ input=x | input=y ]|
%      -------------------------------
       | state=a |    a    |    b    ||
%      -------------------------------
       | state=b |    b    |    b    ||
%      -------------------------------
   ENDTABLE
```

This will translate internally into the LET and COND form shown earlier. Note that the horizontal lines are simply PVS comments.[6]

[6]The LaTeX generation translates these constructs into attractively typeset tables. See the PVS System Guide [9] for details.

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.

```
   TABLE state,    input
%                ----------
                |[ x | y ]|
%        -----------------
         |  a   | a | b ||
%        -----------------
         |  b   | b | b ||
%        -----------------
   ENDTABLE
```

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.

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

And as a horizontal one as follows.

```
  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.

| current mode | Event | New Mode |
|---|---|---|
| $m_1$ | $e_{1,1}$ | $m_{1,1}$ |
| | $e_{1,2}$ | $m_{1,2}$ |
| | $\dots$ | $\dots$ |
| | $e_{1,k_1}$ | $m_{1,k_1}$ |
| $m_2$ | $e_{2,1}$ | $m_{2,1}$ |
| | $e_{2,2}$ | $m_{2,2}$ |
| | $\dots$ | $\dots$ |
| | $e_{2,k_2}$ | $m_{2,k_2}$ |
| $\dots$ | $\dots$ | $\dots$ |
| $m_p$ | $e_{p,1}$ | $m_{p,1}$ |
| | $e_{p,2}$ | $m_{p,2}$ |
| | $\dots$ | $\dots$ |
| | $e_{p,k_p}$ | $m_{p,k_p}$ |

And translate to the following form.

```
TABLE mode
%--------------------------------
     |  m_1 | TABLE event
                    | e_1,1 | m_1,1 ||
                    | e_1,2 | m_1,2 ||
           ...
                    | e_1,k1| m_1,k1||
             ENDTABLE ||
%--------------------------------
     |  m_2 | TABLE event
                    | e_2,1 | m_2,1 ||
                    | e_2,2 | m_2,2 ||
           ...
                    | e_2,k2| m_2,k2||
             ENDTABLE ||
%--------------------------------
        ...
%--------------------------------
     |  m_p | TABLE event
                    | e_p,1 | m_p,1 ||
                    | e_p,2 | m_p,2 ||
           ...
                    | e_p,kp| m_p,kp||
             ENDTABLE ||
%--------------------------------
ENDTABLE
```

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(2) ||
  | y = 0 | DC(0) |   2   |    3  ||
  | y > 0 |   -2  | DC(1) |    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* \| *Datatype* }$_+$ |
| *Theory* | ::= | *Id* [ *TheoryFormals* ] `: THEORY`<br>[ *Exporting* ]<br>`BEGIN`<br>[ *AssumingPart* ]<br>[ *TheoryPart* ]<br>`END` *Id* |
| *TheoryFormals* | ::= | `[` *TheoryFormal*$_{++}$'`,`' `]` |
| *TheoryFormal* | ::= | [ `(` *Importing* `)` ] *TheoryFormalDecl* |
| *TheoryFormalDecl* | ::= | *TheoryFormalType* \| *TheoryFormalConst* |
| *TheoryFormalType* | ::= | *Ids* `:` { `TYPE` \| `NONEMPTY_TYPE` \| `TYPE+` }<br>[ `FROM` *TypeExpr* ] |
| *TheoryFormalConst* | ::= | *IdOps* `:` *TypeExpr* |

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 `IMPORTING`s. 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.

The groups theory below illustrates these concepts. It views a group as a 4-tuple consisting of a type G, an identity element e of G, and operations o[1] and inv. Note the use of the type parameter G in the rest of the formal parameter list. The assuming part provides the group axioms. Any use of the groups theory incurs the obligation to prove all of the ASSUMPTIONs. The body of the groups theory consists of two theorems, which can be proved from the assumptions.

```
groups [G : TYPE,
        e : G,
        o : [G,G->G],
        inv : [G->G] ] : THEORY
  BEGIN
    ASSUMING
      a, b, c: VAR G

      associativity : ASSUMPTION  a o (b o c) = (a o b) o c

      unit : ASSUMPTION  e o a = a AND a o e = a

      inverse : ASSUMPTION inv(a) o a = e AND a o inv(a) = e

    ENDASSUMING

    left_cancellation: THEOREM a o b = a o c IMPLIES b = c

    right_cancellation: THEOREM b o a = c o a IMPLIES b = c

  END groups
```

Figure 6.2: Theory groups

## 6.1 Theory Identifiers

The theory identifier introduces a name for a theory; as described in Section 7, this identifier can be used to help disambiguate references to declarations of the theory.

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, which provides, among other things, the Boolean operators, equality, and the real, rational, integer, and naturalnumber 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.

[1]Recall that o is an infix operator.

## 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, or constants, and `IMPORTING`s may be interspersed. Theory parameters must have unique identifiers. The parameters are ordered, allowing later parameters to refer to earlier ones. This is another form of dependency, akin to dependent types (see Section 4.5). A theory is *instantiated* from within another theory by providing *actual parameters* to substitute for the formals. Actual parameters may occur in `IMPORTING`s, `EXPORTING`s, theory declarations, and names. In each case they are enclosed in braces (`[` and `]`) and separated with commas.

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

```
T [t: TYPE,
   subt: TYPE FROM t
   (IMPORTING orders[subt]) <=: (partial_order?),
   c: subt,
   d: {x:subt | c <= x}]
```

a valid instance has five actual parameters; an example is

```
T[int, {x:nat | 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.

| | | |
|---|---|---|
| *Exporting* | ::= | `EXPORTING` *ExportingNames* [`WITH` *ExportingTheories*] |
| *ExportingNames* | ::= | `ALL` [`BUT` *ExportingName*++ ',' ] |
| | \| | *ExportingName*++ ',' |
| *ExportingName* | ::= | *IdOp* [ `:` { *TypeExpr* \| `TYPE` \| `FORMULA` } ] |
| *ExportingTheories* | ::= | `ALL` \| `CLOSURE` \| *TheoryNames* |
| *Importing* | ::= | `IMPORTING` *TheoryNames* |

Figure 6.3: Importing and Exporting Syntax

### 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 6.3. The `EXPORTING` clause is optional; if omitted, it defaults to

```
EXPORTING ALL WITH ALL
```

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 `AXIOM`s, `LEMMA`s, 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

```
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 `BUT`.

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.[2] 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.

[2] Proofs are not used in completion analysis.

### 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 import the theory name that they abbreviate (see Section 6.4).

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. `IMPORTING`s 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 `IMPORTING`s 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 7 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.

## 6.4 Theory Abbreviations

A theory abbreviation introduces a new name for a theory instance, providing an alternate means for referring to the instance. For example, given the declaration

```
fsets: THEORY = sets[[integer -> integer]]
```

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`. The body of a theory abbreviation must refer to a theory of the current context or the prelude, and the actual parameters must be compatible (in number, kind, and type) with the corresponding theory parameters.

In addition to providing an abbreviation, such declarations do an implicit `IMPORTING` of the theory instance. Theory abbreviations may not be exported.

## 6.5 Assuming Part

The assuming part consists of top-level declarations and `IMPORTING`s. 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 6.4.

The primary purpose of the assuming part is to provide constraints on the use of the theory, by means of `ASSUMPTION`s. These are formula expressing properties that are expected to hold of any instance of the theory. They are generally stated in terms of the formal

| | | |
|---|---|---|
| *AssumingPart* | ::= | `ASSUMING` {*AssumingElement* [ ; ]}$_{+}$ `ENDASSUMING` |
| *AssumingElement* | ::= | *Importing* |
| | \| | *TheoryDecl* |
| | \| | *Assumption* |

Figure 6.4: Assuming Syntax

parameters, and when instantiated they become *assuming* TCCs `OBLIGATION`s which must be discharged. For example, given the theory `groups` above, the importing

```
IMPORTING groups[int, 0, +, -]
```

generates the following obligations

```
% Assuming TCC generated for  groups[int, 0, +, -]
IMPORTING1_TCC1: OBLIGATION
  (FORALL (a: int), (b: int), (c: int):
       +(a, (+(b, c))) = +((+(a, b)), c))

% Assuming TCC generated for  groups[int, 0, +, -]
IMPORTING1_TCC2: OBLIGATION (FORALL (a: int):
       +(0, a) = a AND +(a, 0) = a)

% Assuming TCC generated for  groups[int, 0, +, -]
IMPORTING1_TCC3: OBLIGATION (FORALL (a: int):
       +(-(a), a) = 0 AND +(a, -(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.

## 6.6 Theory Part

The theory part consists of top-level declarations and `IMPORTING`s. 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 6.5.

| | | |
|---|---|---|
| *TheoryPart* | ::= | { *TheoryElement* [ ; ] }+ |
| *TheoryElement* | ::= | *Importing* \| *TheoryDecl* |
| *TheoryDecl* | ::= | *LibDecl* \| *TheoryAbbrDecl* \| *TypeDecl* \| *VarDecl* |
| | \| | *ConstDecl* \| *RecursiveDecl* \| *MacroDecl* \| *InductiveDecl* |
| | \| | *FormulaDecl* \| *Judgement* \| *Conversion* \| *InlineDatatype* |
| | \| | *AutoRewriteDecl* |

Figure 6.5: Theory Part Syntax

# Chapter 7

# 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 7.1.

```
TheoryNames  ::= TheoryName++','
TheoryName   ::= [Id @] Id [Actuals]
Names        ::= Name++','
Name         ::= [Id @] IdOp [Actuals] [. IdOp]
Actuals      ::= [ Actual++',' ]
Actual       ::= Expr | TypeExpr
IdOps        ::= IdOp++','
IdOp         ::= Id | Opsym
Opsym        ::= Binop | Unaryop | IF | TRUE | FALSE | [||] | (||) | {||}
Binop        ::= o | IFF | <=> | IMPLIES | => | WHEN | OR | \/ | AND
               | /\ | & | XOR | ANDTHEN | ORELSE | ^ | + | - | * | /
               | ++ | ~ | ** | // | ^^ | |- | |= | <| | |> | =
               | /= | == | < | <= | > | >= | <<
               | >> | <<= | >>= | # | @@ | ##
Unaryop      ::= NOT | ~ | [] | <> | -
FormulaName  ::= AXIOM | CHALLENGE | CLAIM | CONJECTURE | COROLLARY
               | FACT | FORMULA | LAW | LEMMA | OBLIGATION
               | POSTULATE | PROPOSITION | SUBLEMMA | THEOREM
```

Figure 7.1: Name Syntax

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 52). Theory parameters must be unique across all kinds.

There are three possible forms for names (two for theory names, which appear in `IMPORTING`s, `EXPORTING WITH`s, 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*:

- *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. These forms are listed in order of increasing 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

```
c1: int = c2
```

`c2` has expected type `int`. For most expressions, this is straight-forward, but applications create special problems. For example, in

```
f: FORMULA c1 = c2
```

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 `real` regardless of its context. However, for

[1] Of course, this assumes that `TRUE` has not itself been overloaded.

the purpose of checking for TCCs, it may be treated as having a different type depending on the expected type and the available judgements.

# Chapter 8

# 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 [2]). 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.

```
Datatype        ::=  Id [ TheoryFormals ] : DATATYPE [WITH SUBTYPES Ids]
                     BEGIN
                     [ Importing [ ; ] ]
                     [ AssumingPart ]
                     DatatypePart
                     END Id

InlineDatatype  ::=  Id : DATATYPE [WITH SUBTYPES Ids]
                     BEGIN
                     [ Importing [ ; ] ]
                     [ AssumingPart ]
                     DatatypePart
                     END id

DatatypePart    ::=  { Constructor : IdOp [ : Id ] }+

Constructor     ::=  IdOp [ ( {IdOps : TypeExpr }++',' ) ]
```

Figure 8.1: Datatype Syntax

The syntax for PVS datatypes is given in Figure 8.1. 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 `foo` causes the generation of a new PVS file named `foo_adt.pvs` containing up to three theories as described below. Typechecking an in-line

[1] Enumeration types are actually in-line datatypes—see Section 3.1.4.

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 generated for an in-line datatype may be viewed using the `M-x prettyprint-expanded` command (see the *PVS System Guide* [9]).

## 8.1 A Datatype Example: `stack`

An example of a datatype is `stack`:

```
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 `ord` function is then defined to return `0` on an empty stack and `1` on a nonempty stack.

Then a series of axioms are given. The `stack_empty_extensionality` axiom states that there is only one bottom element of the datatype: `empty`. `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 `pop`ping and `push`ing the same element off and onto a stack results in a stack identical to the original. `stack_top_push` says that if you `push` and element on a stack, you get that same element when you `pop` it back off. `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. Another related axiom `stack_disjointness` is implicitly available but is not output explicitly in the generated theory. In the case of the stacks datatype this axiom would be as follows.

```
stack_disjointness: AXIOM
      (FORALL (stack_var: stack):
         NOT (emptystack?(stack_var) AND nonemptystack?(stack_var)));
```

```
%%% ADT file generated from stack

stack_adt[T: TYPE]: THEORY
 BEGIN

  stack: TYPE

  empty?, nonempty?: [stack -> boolean]

  empty: (empty?)

  push: [[T, stack] -> (nonempty?)]

  top: [(nonempty?) -> T]

  pop: [(nonempty?) -> stack]

  ord(x: stack): upto(1) =
      CASES x OF empty: 0, push(push1_var, push2_var): 1 ENDCASES

  stack_empty_extensionality: AXIOM
    FORALL (empty?_var: (empty?), empty?_var2: (empty?)):
      empty?_var = empty?_var2;

  stack_push_extensionality: AXIOM
    FORALL (nonempty?_var: (nonempty?), nonempty?_var2: (nonempty?)):
      top(nonempty?_var) = top(nonempty?_var2) AND
       pop(nonempty?_var) = pop(nonempty?_var2)
       IMPLIES nonempty?_var = nonempty?_var2;

  stack_push_eta: AXIOM
    FORALL (nonempty?_var: (nonempty?)):
      push(top(nonempty?_var), pop(nonempty?_var)) = nonempty?_var;

  stack_top_push: AXIOM
    FORALL (push1_var: T, push2_var: stack):
      top(push(push1_var, push2_var)) = push1_var;

  stack_pop_push: AXIOM
    FORALL (push1_var: T, push2_var: stack):
      pop(push(push1_var, push2_var)) = push2_var;
```

Figure 8.2: Theory `stack_adt` (continues)

For datatypes with several constructors[2] such an axiom could grow prohibitively large, slowing down the generation of datatypes as well as some prooftime operations. Thus the explicit generation of this axiom is suppressed. Since this axiom is built in to the prover when the `stacks` theory is typechecked, there is no need for this axiom.

[2]For example, enumeration types with a lot of elements.

```
stack_inclusive: AXIOM
  FORALL (stack_var: stack): empty?(stack_var) OR nonempty?(stack_var);

stack_disjoint: AXIOM
  FORALL (stack_var: stack):
    NOT (empty?(stack_var) AND nonempty?(stack_var));

stack_induction: AXIOM
  FORALL (p: [stack -> boolean]):
    (p(empty) AND
      (FORALL (push1_var: T, push2_var: stack):
         p(push2_var) IMPLIES p(push(push1_var, push2_var))))
     IMPLIES (FORALL (stack_var: stack): p(stack_var));

every(p: PRED[T])(a: stack):  boolean =
    CASES a
      OF empty: TRUE,
         push(push1_var, push2_var): p(push1_var) AND every(p)(push2_var)
      ENDCASES;

every(p: PRED[T], a: stack):  boolean =
    CASES a
      OF empty: TRUE,
         push(push1_var, push2_var): p(push1_var) AND every(p, push2_var)
      ENDCASES;

some(p: PRED[T])(a: stack):  boolean =
    CASES a
      OF empty: FALSE,
         push(push1_var, push2_var): p(push1_var) OR some(p)(push2_var)
      ENDCASES;

some(p: PRED[T], a: stack):  boolean =
    CASES a
      OF empty: FALSE,
         push(push1_var, push2_var): p(push1_var) OR some(p, push2_var)
      ENDCASES;

subterm(x, y: stack):  boolean =
    x = y OR
     CASES y
       OF empty: FALSE, push(push1_var, push2_var): subterm(x, push2_var)
       ENDCASES;
```

Figure 8.3: Theory `stack_adt` (continues)

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

```
<<:  (well_founded?[stack]) =
    LAMBDA (x, y: stack):
      CASES y
        OF empty: FALSE,
           push(push1_var, push2_var): x = push2_var OR x << push2_var
        ENDCASES;

stack_well_founded: AXIOM well_founded?[stack](<<);

reduce_nat(empty?_fun: nat, nonempty?_fun: [[T, nat] -> nat]):
[stack -> nat] =
    LAMBDA (stack_adtvar: stack):
      LET red: [stack -> nat] = reduce_nat(empty?_fun, nonempty?_fun) IN
        CASES stack_adtvar
          OF empty: empty?_fun,
             push(push1_var, push2_var):
               nonempty?_fun(push1_var, red(push2_var))
          ENDCASES;

REDUCE_nat(empty?_fun: [stack -> nat],
           nonempty?_fun: [[T, nat, stack] -> nat]):
 [stack -> nat] =
    LAMBDA (stack_adtvar: stack):
      LET red: [stack -> nat] = REDUCE_nat(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)
          ENDCASES;

reduce_ordinal(empty?_fun: ordinal,
               nonempty?_fun: [[T, ordinal] -> 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,
             push(push1_var, push2_var):
               nonempty?_fun(push1_var, red(push2_var))
          ENDCASES;
```

Figure 8.4: Theory `stack_adt` (continues)

2. if $p$ holds for some stack then $p$ holds for the result of `push`ing 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

```
  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)
            ENDCASES;
 END stack_adt

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))
        ENDCASES;

  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))
        ENDCASES;
 END stack_adt_map
```

Figure 8.5: Theory `stack_adt_map`

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 `<<` predicate is the strict (irreflexive) `subterm` predicate. Thus for all stacks $s$, `subterm`$(s, s)$ holds, but for no stack $s$ does `<<`$(s, s)$ hold. An alternative equivalent definition of `<<` is as follows:

```
  <<(x: stack, y: stack): boolean = subterm(x,y) 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 `<<`, 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 func-

```
stack_adt_reduce[T: TYPE, range: TYPE]: THEORY
 BEGIN

  IMPORTING stack_adt[T]

  reduce(empty?_fun: range, nonempty?_fun: [[T, range] -> range]):
  [stack -> range] =
      LAMBDA (stack_advar: stack):
        LET red: [stack -> range] = reduce(empty?_fun, nonempty?_fun) IN
          CASES stack_adtvar
            OF empty: empty?_fun,
               push(push1_var, push2_var):
                 nonempty?_fun(push1_var, red(push2_var))
            ENDCASES;

  REDUCE(empty?_fun: [stack -> range],
         nonempty?_fun: [[T, range, stack] -> range]):
   [stack -> range] =
      LAMBDA (stack_adtvar: stack):
        LET red: [stack -> range] = REDUCE(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)
            ENDCASES;
 END stack_adt_reduce
```

Figure 8.6: Theory `stack_adt_reduce`

tions 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 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.

The third and final theory in file `stack_adt.pvs` generated from `stack_pvs` is `stack_adt_reduce`. This theory provides a generalized version of `reduce_nat` and `REDUCE_nat`. The theory `stack_adt_reduce` 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.

## 8.2 Datatype Details

In general, a datatype declaration has the form

$$
\begin{array}{l}
\texttt{adt: DATATYPE WITH SUBTYPES } \texttt{S}_1\texttt{, ..., S}_n\\
\quad\texttt{BEGIN}\\
\quad\quad\texttt{cons}_1\texttt{(acc}_{11}\texttt{: T}_{11}\texttt{, ..., acc}_{1n_1}\texttt{: T}_{1n_1}\texttt{): rec}_1\texttt{ : S}_{i_1}\\
\quad\quad\vdots\\
\quad\quad\texttt{cons}_m\texttt{(acc}_{m1}\texttt{: T}_{m1}\texttt{, ..., acc}_{1n_m}\texttt{: T}_{1n_m}\texttt{): rec}_m\texttt{ : S}_{i_m}\\
\quad\texttt{END adt}
\end{array}
$$

where the $\texttt{cons}_i$ are the *constructors*, the $\texttt{acc}_{ij}$ are the *accessors*, the $\texttt{T}_{ij}$ are type expressions, and the $\texttt{rec}_i$ are *recognizers*. Each line is referred to as a *constructor specification*. There are a number of restrictions enforced on constructor specifications:

- The datatype identifier may not be used for a recognizer, accessor, or subtype: ($\texttt{adt} \not\equiv \texttt{rec}_i$ for all $i$, $\texttt{adt} \not\equiv \texttt{acc}_{ij}$ for all $i$ and $j$, and $\texttt{adt} \not\equiv \texttt{S}_i$ for all $i$).
- The subtype names must be unique: ($i \neq j \Rightarrow \texttt{S}_i \not\equiv \texttt{S}_j$)
- Each subtype name must be used at least once.
- The constructor names must be unique: ($i \neq j \Rightarrow \texttt{cons}_i \not\equiv \texttt{cons}_j$).
- The recognizer names must be unique: ($i \neq j \Rightarrow \texttt{rec}_i \not\equiv \texttt{rec}_j$).
- No identifier may be used as both a constructor and a recognizer: ($\texttt{cons}_i \not\equiv \texttt{rec}_j$ forall $i$ and $j$).
- Duplicate accessor identifiers are not allowed within a single constructor specification: ($j \neq k \Rightarrow \texttt{acc}_{ij} \not\equiv \texttt{acc}_{ik}$).

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

- $\tau \equiv \texttt{T}$.

- $\tau \equiv \{x : \tau' | p(x)\}$ and the occurrence `T` is positive in $\tau'$.
- $\tau \equiv [\tau_1 \rightarrow \tau_2]$ and the occurrence `T` is positive in $\tau_2$. For example, `T` occurs positively in `sequence[T]` where `sequence[T]` is defined in the PVS prelude as the function type `[nat -> T]`.
- $\tau \equiv [\tau_1, \ldots, \tau_n]$ and the occurrence `T` is positive in some $\tau_i$.
- $\tau \equiv [\# \; l_1 : \tau_1, \ldots, l_n : \tau_n \; \#]$ and the occurrence `T` is positive in some $\tau_i$.
- $\tau \equiv$ *datatype*$[\tau_1, \ldots, \tau_n]$, where *datatype* is a previously defined datatype and the occurrence `T` is positive in $\tau_i$, where $\tau_i$ is a *positive parameter* of *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 `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.*,

  $\mathtt{acc}_{ij}\mathtt{(cons}_i\mathtt{(e}_{i1}\mathtt{, \ldots,\ e}_{im_i}\mathtt{)\ =\ e}_{ij}$
- 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 as the prover checks for this directly.
- Two induction schemes are provided for proving properties of the datatype.
- If 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.
- The recursive combinators `reduce_nat`, `REDUCE_nat`, `reduce_ordinal`, and `REDUCE_ordinal` are defined. These provide a means for defining notions like the size or depth of a datatype term.
- If all type parameters are positive and no accessor type involves a subtype of the datatype,[3] 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 `stack` 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 3.1.5.

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 $\{s : \texttt{stack[}T\texttt{]} | \texttt{every}(p)(s)\}$. 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 `dt[nat, nat, 0]` is a subtype of `dt[int, nat, 0]`, but is not a subtype of `dt[nat, int, 0]`, nor is it a subtype of `dt[nat, nat, 1]`.

More complex datatypes lead to correspondingly more complex declarations; for example, in the following 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:

[3] These constraints are too strong, and may be modified in the future.

```
every(p: PRED[t1])(a1: adt1): boolean =
  CASES a1 OF
    bottom: TRUE,
    c1(c11_var, c12_var): p(c11_var),
    c2(c21_var, c22_var, c23_var):
        every(p)(c21_var)
            AND every(every(p))(c22_var)
              AND every[adt1](every(p))(c23_var),
    c3(c31_var, c32_var, c33_var):
        (FORALL (x1: list[int]): every(p)(c31_var(x1)))
            AND
          (every(p)(a(c32_var))
               AND (FORALL (x: int): every(p)(b(c32_var)(x))))
              AND every(p)(c33_var`1)
                AND
              (FORALL (x: set[int]):
                 every(p)(c33_var`2(x)))
    ENDCASES;
```

Note that this is only defined for predicates over `t1`, since the occurrence of `t2` in the constructor specification for `c2` is not positive.

Like record types, constructor selectors may be dependent

**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:

$$\begin{array}{lcl} T & ::= & B \mid T \to T \\ E & ::= & x \mid \lambda x : T.E \mid E(E) \end{array}$$

This is difficult to express using datatypes without subtypes, but is reasonably straightforward with them:[4]

```
tlc: DATATYPE WITH SUBTYPES typ, expr
 BEGIN
 base_type(n:nat): base_type? : typ
 fun_type(dom, ran: typ): fun_type? : typ
 expr_var(n:nat): expr_var? : expr
 lambda_expr(lvar:(expr_var?), ltype: typ, lexpr: expr)
                            : lambda_expr? : expr
 application(fun, arg: expr): application? : expr
 END tlc
```

In addition to the usual generated declarations, this generates

```
  typ((x: tlc)): boolean = base_type?(x) OR fun_type?(x);
  typ: TYPE = {x: tlc | base_type?(x) OR fun_type?(x)}
  expr((x: tlc)): boolean =
     expr_var?(x) OR lambda_expr?(x) OR application?(x);
  expr: TYPE =
     {x: tlc | expr_var?(x) OR lambda_expr?(x) OR application?(x)}
```

[4] `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.

## 8.3 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

  *ExportingName* ::= *IdOp* [ `:` { *TypeExpr* | `TYPE` | `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 `{}`.

## Specification

| | | |
|---|---|---|
| *Specification* | ::= | { *Theory* \| *Datatype* }+ |
| *Theory* | ::= | *Id* [ *TheoryFormals* ] `:` `THEORY`<br>[ *Exporting* ]<br>`BEGIN`<br>[ *AssumingPart* ]<br>[ *TheoryPart* ]<br>`END` *Id* |
| *TheoryFormals* | ::= | `[` *TheoryFormal*++ ',' `]` |
| *TheoryFormal* | ::= | [ `(` *Importing* `)` ] *TheoryFormalDecl* |
| *TheoryFormalDecl* | ::= | *TheoryFormalType* \| *TheoryFormalConst* |
| *TheoryFormalType* | ::= | *Ids* `:` { `TYPE` \| `NONEMPTY_TYPE` \| `TYPE+` }<br>[ `FROM` *TypeExpr* ] |
| *TheoryFormalConst* | ::= | *IdOps* `:` *TypeExpr* |

## Importings and Exportings

| | | |
|---|---|---|
| *Exporting* | ::= | `EXPORTING` *ExportingNames* [ `WITH` *ExportingTheories* ] |
| *ExportingNames* | ::= | `ALL` [ `BUT` *ExportingName*++ ',' ] |
| | \| | *ExportingName*++ ',' |
| *ExportingName* | ::= | *IdOp* [ `:` { *TypeExpr* \| `TYPE` \| `FORMULA` } ] |
| *ExportingTheories* | ::= | `ALL` \| `CLOSURE` \| *TheoryNames* |
| *Importing* | ::= | `IMPORTING` *TheoryNames* |

## Assumings

| | | |
|---|---|---|
| *AssumingPart* | ::= | `ASSUMING` { *AssumingElement* [ `;` ] }+ `ENDASSUMING` |
| *AssumingElement* | ::= | *Importing* |
| | \| | *TheoryDecl* |
| | \| | *Assumption* |

## Theory Part

| | | |
|---|---|---|
| *TheoryPart* | ::= | { *TheoryElement* [ ; ] }$_+$ |
| *TheoryElement* | ::= | *Importing* \| *TheoryDecl* |
| *TheoryDecl* | ::= | *LibDecl* \| *TheoryAbbrDecl* \| *TypeDecl* \| *VarDecl* |
| | \| | *ConstDecl* \| *RecursiveDecl* \| *MacroDecl* \| *InductiveDecl* |
| | \| | *FormulaDecl* \| *Judgement* \| *Conversion* \| *InlineDatatype* |
| | \| | *AutoRewriteDecl* |

### Declarations

*LibDecl* ::= *Ids* : LIBRARY [ = ] *String*

*TheoryAbbrDecl* ::= *Ids* : THEORY = *TheoryName*

*TypeDecl* ::= *Id* [ { , *Ids* } | *Bindings* ] :
{ TYPE | NONEMPTY_TYPE | TYPE+ }
[ { = | FROM } *TypeExpr* [ CONTAINING *Expr* ] ]

*VarDecl* ::= *IdOps* : VAR *TypeExpr*

*ConstDecl* ::= *IdOp* [ { , *IdOps* } | *Bindings*+ ] : *TypeExpr* [ = *Expr* ]

*RecursiveDecl* ::= *IdOp* [ { , *IdOps* } | *Bindings*+ ] : RECURSIVE
*TypeExpr* = *Expr* MEASURE *Expr* [ BY *Expr* ]

*MacroDecl* ::= *IdOp* [ { , *IdOps* } | *Bindings*+ ] : MACRO
*TypeExpr* = *Expr*

*InductiveDecl* ::= *IdOp* [ { , *IdOps* } | *Bindings*+ ] : INDUCTIVE
*TypeExpr* = *Expr*

*Assumption* ::= *Ids* : ASSUMPTION *Expr*

*FormulaDecl* ::= *Ids* : *FormulaName* *Expr*

*Judgement* ::= *SubtypeJudgement* | *ConstantJudgement*

*SubtypeJudgement* ::= [ *IdOp* : ] JUDGEMENT *TypeExpr*++ ’,’ SUBTYPE_OF *TypeExpr*

*ConstantJudgement* ::= [ *IdOp* : ] JUDGEMENT *ConstantReference*++ ’,’
HAS_TYPE *TypeExpr*

*ConstantReference* ::= *Number* | { *Name* *Bindings** }

*Conversion* ::= { CONVERSION | CONVERSION+ | CONVERSION- }
{ *Name* [ : *TypeExpr* ] }++ ’,’

*AutoRewriteDecl* ::= { AUTO_REWRITE | AUTO_REWRITE+ | AUTO_REWRITE- }
*RewriteName*++ ’,’

*RewriteName* ::= *Name* [ ! [ ! ] ] [ : { *TypeExpr* | *FormulaName* } ]

*Bindings* ::= ( *Binding*++ ’,’ )

*Binding* ::= *TypedId* | { ( *TypedIds* ) }

*TypedIds* ::= *IdOps* [ : *TypeExpr* ] [ | *Expr* ]

*TypedId* ::= *IdOp* [ : *TypeExpr* ] [ | *Expr* ]

## Type Expressions

| | | |
|---|---|---|
| *TypeExpr* | ::= | *Name* |
| | \| | *EnumerationType* |
| | \| | *Subtype* |
| | \| | *TypeApplication* |
| | \| | *FunctionType* |
| | \| | *TupleType* |
| | \| | *RecordType* |
| *EnumerationType* | ::= | { *IdOps* } |
| *Subtype* | ::= | { *SetBindings* `\|` *Expr* } |
| | \| | `(` *Expr* `)` |
| *TypeApplication* | ::= | *Name Arguments* |
| *FunctionType* | ::= | [`FUNCTION` \| `ARRAY`] <br> `[` { [*IdOp* `:`] *TypeExpr* }$_{++}$ ’`,`’ `->` *TypeExpr* `]` |
| *TupleType* | ::= | `[` { [*IdOp* `:`] *TypeExpr* }$_{++}$ ’`,`’ `]` |
| *RecordType* | ::= | `[#` *FieldDecls*$_{++}$ ’`,`’ `#]` |
| *FieldDecls* | ::= | *Ids* `:` *TypeExpr* |

### Expressions

```
Expr  ::=  Number
        |  String
        |  Name
        |  Id ! Number
        |  Expr Arguments
        |  Expr Binop Expr
        |  Unaryop Expr
        |  Expr ` { Id | Number }
        |  ( Expr++ ',' )
        |  (: 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
```

## Expressions (continued)

```
IfExpr          ::=  IF Expr THEN Expr
                     { ELSIF Expr THEN Expr } * ELSE Expr ENDIF

BindingExpr     ::=  BindingOp LambdaBindings : Expr

BindingOp       ::=  LAMBDA | FORALL | EXISTS | { IdOp ! }

LambdaBindings  ::=  LambdaBinding [ [,] LambdaBindings ]

LambdaBinding   ::=  IdOp | Bindings

SetBindings     ::=  SetBinding [ [,] SetBindings ]

SetBinding      ::=  { IdOp [ : TypeExpr ] } | Bindings

Assignment      ::=  AssignArgs { := | |-> } Expr

AssignArgs      ::=  Id [ ! Number ]
                 |   Number
                 |   AssignArg+

AssignArg       ::=  ( Expr++',' )
                 |   ` Id
                 |   ` Number

Selection       ::=  IdOp [ ( IdOps ) ] : Expr

TableExpr       ::=  TABLE [ Expr ] [ , Expr ]
                     [ ColHeading ]
                     TableEntry+ ENDTABLE

ColHeading      ::=  |[ Expr { | { Expr | ELSE } }+ ]|

TableEntry      ::=  { | [ Expr | ELSE ] }+ ||

LetBinding      ::=  { LetBind | ( LetBind++',' ) } = Expr

LetBind         ::=  IdOp Bindings* [ : TypeExpr ]

Arguments       ::=  ( Expr++',' )
```

## Names

```
TheoryNames  ::=  TheoryName++ ','
TheoryName   ::=  [ Id @ ] Id [ Actuals ]
Names        ::=  Name++ ','
Name         ::=  [ Id @ ] IdOp [ Actuals ] [ . IdOp ]
Actuals      ::=  [ Actual++ ',' ]
Actual       ::=  Expr | TypeExpr
IdOps        ::=  IdOp++ ','
IdOp         ::=  Id | Opsym
Opsym        ::=  Binop | Unaryop | IF | TRUE | FALSE | [||] | (||) | {||}
Binop        ::=  o | IFF | <=> | IMPLIES | => | WHEN | OR | \/ | AND
               |  /\ | & | XOR | ANDTHEN | ORELSE | ^ | + | - | * | /
               |  ++ | ~ | ** | // | ^^ | |- | |= | <| | |> | =
               |  /= | == | < | <= | > | >= | <<
               |  >> | <<= | >>= | # | @@ | ##
Unaryop      ::=  NOT | ~ | [] | <> | -
FormulaName  ::=  AXIOM | CHALLENGE | CLAIM | CONJECTURE | COROLLARY
               |  FACT | FORMULA | LAW | LEMMA | OBLIGATION
               |  POSTULATE | PROPOSITION | SUBLEMMA | THEOREM
```

## Identifiers

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

### Datatypes

| | | |
|---|---|---|
| *Datatype* | ::= | *Id* [ *TheoryFormals* ] `: DATATYPE` [`WITH SUBTYPES` *Ids*]<br>`BEGIN`<br>[ *Importing* [ `;` ] ]<br>[ *AssumingPart* ]<br>*DatatypePart*<br>`END` *Id* |
| *InlineDatatype* | ::= | *Id* `: DATATYPE` [`WITH SUBTYPES` *Ids*]<br>`BEGIN`<br>[ *Importing* [ `;` ] ]<br>[ *AssumingPart* ]<br>*DatatypePart*<br>`END` *id* |
| *DatatypePart* | ::= | { *Constructor* `:` *IdOp* [`:` *Id*] }$_+$ |
| *Constructor* | ::= | *IdOp* [ `(` { *IdOps* `:` *TypeExpr* }$_{++}$ ',' `)` ] |

# Bibliography


[1] Michael J. Beeson. *Foundations of Constructive Mathematics*. Ergebnisse der Mathematik und ihrer Grenzgebiete; 3. Folge · Band 6. Springer-Verlag, 1985. 4

[2] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979. 71

[3] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In Carroll Morgan and J. C. P. Woodcock, editors, *Proceedings of the Third Refinement Workshop*, pages 51–69. Springer-Verlag Workshops in Computing, 1990. 42

[4] William M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–1291, September 1990. 4

[5] Chris George. The RAISE specification language: A tutorial. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 552 of *Lecture Notes in Computer Science*, pages 238–319, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag. Volume 2: Tutorials. 4

[6] Constance Heitmeyer, Bruce Labaw, and Daniel Kiskis. Consistency checks for SCR-style requirements specifications. Technical report, Naval Research Laboratory, Washington DC, September 1994. In press. 52

[7] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1990. 4

[8] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? SRC Research Report 147, Digital Systems Research Center, Palo Alto, CA, May 1997. Available at `http://www.research.digital.com/SRC`. 3

[9] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. 1, 34, 39, 55, 72

[10] David Lorge Parnas. Tabular representation of relations. Technical Report CRL Report 241, McMaster University, Hamilton, Canada, TRIO (Telecommunication Research Institute of Ontario), October 1992. 52

[11] D. S. Scott. Identity and existence in intuitionistic logic. In *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics*, pages 660–696. Springer, 1979. 4

[12] N. Shankar and S. Owre. *The Formal Semantics of PVS*. Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997. 3

[13] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. 1, 11, 24, 25

# Index