The PVS Specification Language
(Beta Release)

S. Owre, N. Shankar and J. M. Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025
Phone: (415) 859-5272
{Owre,Shankar,Rushby}@csl.sri.com

April 12, 1993

Contents

List of Figures

1 Introduction

PVS is a *P*rototype *V*erification *S*ystem for specifying and verifying digital systems. 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 User Guide* [9] is to be consulted for information on how to use the system to develop specifications and proofs. The *PVS Proof Checker* manual [11] 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, there are two appendices: Appendix A provides the grammar of the language, and Appendix B lists the built-in theories of PVS, referred to as the *prelude*.

Summary of the PVS Specification 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 the *base* types (Booleans, rationals, integers, natural numbers, etc.) using the function, record, and tuple type constructions. For example, the type of a predicate on 2-tuples of integers would be `[[int, int] -> bool]`. 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. 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 rationals can be written as {`x:rational | 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 rationals is given a type where the denominator is constrained to be a nonzero rational. 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.

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 the constraints on *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 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. 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. Functions can represent either *total* or *partial* maps. A total map from domain A to range B maps each element of A to some element of B, whereas a partial map only maps some of the elements of A to elements of B. Most traditional logics build in the assumption that functions represent total maps. Partial functions arise quite naturally in specifications. For example, the division operation is undefined on a zero denominator and the operation of popping a stack is undefined on an empty stack.

Some recent logics, notably those of VDM [7], LUTINS [4], RAISE [5], Beeson [1] and Scott [10], 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.

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 in the rest of this manual. 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 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.

The first line introduces a theory named `stacks` which is parameterized by a type `t` (the *formal parameter* of `stacks`). The uninterpreted type `stack` is declared, and the constant `empty` and variable `s` are declared to be of type `stack`. The defined predicate `nonemptystack?` is then declared on elements of type `stack`; it is defined in terms of a lambda expression, and is TRUE for a given `stack` element iff[1] 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. When used as a type, the form `(P)` where `P` is a predicate on type `T` is equivalent to the type `{x:T | P(x)}` when `x` does not occur free in `P`.

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 acces-

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

```
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) IFF 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: Theory stacks

sors. Finally, there is the theorem pop2push2, which 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). A TCC is a formula that must be proved in order that the expression be type correct. In this case, the TCC is

```
  pop2push2_TCC1: OBLIGATION
    (FORALL (s: stack), (y: t), (x: t):
      nonemptystack?(pop(push(x, push(y, s)))))
```

The system keeps track of all such obligations and will flag the unproved ones during proof chain analysis (see the proof checker manual [11]).

Parameterized theories such as stacks introduce theory schemas, where the type t may be instantiated with any other 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 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. More details can be found in Section 7. In this example, all of the declared entities of the imported theory are available; if it is desirable to hide some of the internals of a theory, then the `EXPORTING` clause may be used (see Section 6).

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 any of 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.

id	⇒	*letter* ⟨ *ident-char* ⟩*
ident-char	⇒	*letter* | *digit* | _ | ?
letter	⇒	*A* | ... | *Z* | *a* | ... | *z*
digit	⇒	*0* | ... | *9*

Figure 2: Lexical Syntax

The reserved words are shown in Figure 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 meanings of these words is given in the appropriate sections; they are collected here for reference.

The special symbols are listed in Figure 4. All of these symbols are separators; they separate identifiers, numbers, and reserved words.

ALL	COROLLARY	FALSE	LAW	RECURSIVE
AND	DATATYPE	FORALL	LEMMA	SOME
ARRAY	ELSE	FORMULA	LET	SUBLEMMA
ASSUMING	ELSIF	FROM	MEASURE	THEN
ASSUMPTION	END	FUNCTION	NOT	THEOREM
AXIOM	ENDASSUMING	IF	O	THEORY
BEGIN	ENDCASES	IFF	OBLIGATION	TRUE
BUT	ENDIF	IMPLIES	OF	TYPE
CASES	EPSILON	IMPORTING	OR	VAR
CLAIM	EXISTS	IN	PLET	WHEN
CONJECTURE	EXPORTING	LAM	POSTULATE	WHERE
CONTAINING	FACT	LAMBDA	PROPOSITION	WITH

Figure 3: PVS Reserved Words

&	(	)	*	+	,
-	->	.	/	/=	:
:=	;	<	<=	<=>	=
=>	>	>=	|	[	]
[#	#]	{	}	(#	#)
\/	/\	~	==	[]	<>
^					

Figure 4: 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.

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, and formulas. 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.[2]

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 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 proof checker manual [11]). Variable, constant, and recursive definition declarations are of kind *expr*; these may be referenced in expressions and actual parameters. Newly introduced names need only be unique within a kind, as there is no way, for example, to use an expr where a type is expected.[3]

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
```

[2]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.

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

`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 5 gives the grammar for declarations.

simple-type-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^{+}_{,}$: `TYPE` [`FROM` *type-expr*]
simple-const-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^{+}_{,}$: *type-expr*
theory-abbr	$\Rightarrow$	*id* : `THEORY` `=` *theory-name*
type-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^{+}_{,}$: `TYPE` [*type-def*]
type-def	$\Rightarrow$	`=` *enum-type* $\mid$ `=` *type-expr* $\mid$ `FROM` *type-expr*
var-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^{+}_{,}$: `VAR` *type-expr*
const-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^{+}_{,}$ [*formals*] : *type-expr* [*const-value*]
def-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^{+}_{,}$ [*formals*] : `RECURSIVE` *type-expr* `=` *expr* `MEASURE` *expr*
assumption	$\Rightarrow$	*id* : `ASSUMPTION` *formula-def*
formula	$\Rightarrow$	*id* : *formula-name* *formula-def*
ids-spec	$\Rightarrow$	$\langle$ *idop* $\rangle^{+}_{,}$ $\mid$ $\langle$ *idop* *formals* $\rangle$
formals	$\Rightarrow$	`(` $\langle$ *formal* $\rangle^{+}_{,}$ `)`
formal	$\Rightarrow$	$\langle$ *idop* $\rangle^{+}_{,}$ [: *type-expr*]
	$\mid$	`(` $\langle$ *idop* $\rangle^{+}_{,}$: *type-expr* `)`
	$\mid$	`(` *idop* [: *type-expr*] `|` *expr* `)`

Figure 5: Declarations Syntax

Type Declarations

Type declarations are used to introduce new type names and to provide abbreviations for type expressions. There are four kinds of type declaration:

- *uninterpreted type declarations*
- *uninterpreted subtype declarations*
- *interpreted type declarations*
- *enumeration type declarations*

These type declarations introduce *type names* that may be referenced in type expressions (see Section 4).

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 assumptions made on an uninterpreted type `T` are that it is nonempty[4] and disjoint from all other types, except for subtypes of `T`. For example,

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

introduces three new nonempty pairwise disjoint types. If desired, further constraints may be put on these types by means of `AXIOM`s (see Section 3).

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.

Uninterpreted Subtype Declarations

Uninterpreted subtype declarations are of the form

```
s: TYPE FROM t
```

This introduces an uninterpreted *subtype* `s` of the *supertype* `t`, which must be nonempty. This has the same meaning as

```
s_pred: [t -> bool]
s: TYPE = (s_pred)
s_nonempty: AXIOM EXISTS (x:t) : s_pred(x)
```

in which a new predicate is introduced in the first line, `s` is given as a *predicated* subtype in the second line[5], and the last line ensures that `s` is nonempty. The only difference is that the axiom is implicit in the former case, and won't need to be invoked when proving theorems. No other assumptions are made about uninterpreted subtypes; in particular, they are not necessarily disjoint unless the corresponding supertypes are disjoint.

Interpreted Type Declarations

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

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

introduces the type name `intfun` as an abbreviation for the type of functions with integer domain and range. In this case,[6] any type expression `T` involving `intfun` is equivalent to the type expression obtained by substituting `[int -> int]` for `intfun`

[4]The assumption that an uninterpreted type or subtype is nonempty only comes into effect when a constant is declared that involves the type; without this assumption there would be an obligation to show that the type was nonempty, which could only be discharged by including an axiom about the type.

[5]This is described in Section 4

[6]Since PVS uses *structural equivalence* instead of *name equivalence*, see Section 4.

in T. The available type expressions include subtypes, function types, tuple types, and record types. These are described in Section 4.

Enumeration Type Declarations

Enumeration type declarations are of the form

```
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 (defined in Section 4). The advantage of treating them as datatypes is that the necessary axioms are automatically generated, and the prover has some built-in facilities for handling datatypes.

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. 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:t): p(x)) AND q(x))
```

The occurrence of x as an argument to p is of type t, shadowing the one of type int.

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). 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

AXIOMS. 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), obtaining

```
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}
```

Recursive Definitions

Recursive definitions are treated as constant declarations, except that the defining expression is required, and a *measure* must be provided. 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 `nat` or `ordinal`. 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. This definition generates a *termination type correctness condition* (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 > x - 1)
```

A termination TCC is generated for each recursive occurrence of the defined entity within the body of the definition. It is obtained by applying the measure to the arguments of the recursive call and comparing that to the measure applied to the original arguments. The *context* of the occurrence is included in the TCC; in this example the occurrence is within the `ELSE` part of an `IF-THEN-ELSE` so the negated condition is an antecedent to the proof obligation.

```
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 6: Theory `f91`

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 6 the termination TCC is

```
f91_TCC6: OBLIGATION
  (FORALL (v: [i: nat -> {j: nat | IF i>100 THEN j=i-10 ELSE j=91 ENDIF}]),
          i:
    NOT i > 100
```

```
    IMPLIES (IF v(i + 11) > 101 THEN 0 ELSE 101 - v(i + 11) ENDIF)
      < (IF i > 101 THEN 0 ELSE 101 - i ENDIF))
```

where the inner calls to `f91` have been replaced by the higher-order variable `v`. 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.

```
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 7: 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 ε_0. The definition of `ordinal` is in Section B. This is primarily useful in handling lexicographical orderings. For example, in the definition of the Ackerman function in Figure 7,[7] 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 the `ordinal` less than, not the `numbers` one.

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

Formula Declarations

Formula declarations introduce *axioms*, *assumptions*, *theorems*, and *obligations*. The identifier associated with the declaration may be referenced during proofs (see the proof checker manual [11]). 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. Theorems may be introduced with any of the keywords `CLAIM`, `CONJECTURE`, `COROLLARY`, `FACT`, `FORMULA`, `LAW`, `LEMMA`, `PROPOSITION`, `SUBLEMMA`, or `THEOREM`.

Assumptions are only allowed in assuming clauses (see Section 6). 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.[8] 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.

[8] 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))`.

4 Types

PVS specifications are strongly typed; every expression must have 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*. Section 3 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*, *predicate types*, *tuple types*, and *record types*. There are also provisions for creating abstract datatypes that are defined in terms of the types given here. See Section 8 for more on datatypes.

Type expressions may appear in theory parameters, type declarations, variable declarations, constant declarations, and recursive definitions. In addition, they may appear in certain expressions (coercions and local bindings, see Sections 5, and 5, respectively), and as actual parameters in names (Section 7). 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.

type-expr	⇒	*type-name* | *subtype* | *function-type*
	|	*tuple-type* | *record-type*
type-name	⇒	*name*
subtype	⇒	*expr-as-type* | *set-subtype*
expr-as-type	⇒	`(` *expr* `)`
set-subtype	⇒	{ *expr* `:` *type-expr* `|` *expr* } [`CONTAINING` *expr*]
function-type	⇒	[⟨ `FUNCTION` | `ARRAY` ⟩] `[` ⟨ *type-expr* | *id* `:` *type-expr* ⟩$^{+}_{,}$ `->` *type-expr* `]`
enumeration-type	⇒	{ ⟨ *id* ⟩$^{+}_{,}$ }
tuple-type	⇒	`[` ⟨ *type-expr* | *id* `:` *type-expr* ⟩$^{+}_{,}$ `]`
record-type	⇒	`[#` ⟨ *field-decl* ⟩$^{+}_{,}$ `#]`
field-decl	⇒	⟨ *id* ⟩$^{+}_{,}$ `:` *type-expr*

Figure 8: Type Expression Syntax

Subtypes

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

```
t: TYPE = {x: s | p(x)}
```

where `p` is a predicate on the type `s`.[9] This has the usual set-theoretical meaning, since types in PVS may be 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.

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

```
nat: TYPE = {x:integer | x >= 0}
```

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 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*. 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 *nonempty* 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 uses that theory and declares a constant of that type will generate the nonempty TCC. Recall that uninterpreted subtypes are assumed to be nonempty if a constant declaration raises the possibility of a closed expression of this type. Once a type is assumed to be nonempty, it will never cause the generation of a nonempty TCC.

A subtype may be guaranteed nonempty by providing a *witness*, which causes the TCC to be triggered off the subtype expression instead of some constant involving the subtype. This is done using the `CONTAINING` clause of a type declaration, 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[10]

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

[10] This TCC will be proved automatically by the PVS system; see the `typecheck-prove` command in the User Guide.

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

The second TCC associated with subtypes involves 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 `[number, nonzeronumber -> number]`. 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 `nonzeronumber`. The typechecker thus generates a 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,

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`

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.

Function Types

Function types have three equivalent forms:

- `[`t_1`, ..., `t_n` -> t]`
- `FUNCTION[`t_1`, ..., `t_n` -> t]`
- `ARRAY[`t_1`, ..., `t_n` -> t]`

where each t_i is a type expression. An element of this type is simply a function whose domain is the sequence of types $t_1, \ldots, t_n$, and whose range is `t`. A function type is empty if the range is empty and no type of the domain is.[11] There is no difference in meaning between these three forms; they are provided to support different

[11]Nonempty TCCs generated for function types are more restrictive than necessary, in that the TCC will ensure that the range is nonempty, whereas function types are also nonempty if both the domain and the range are empty (such types contain the empty function).

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 shorthand for `[t -> bool]`.[12] 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.

In the current version of PVS, a function type [t_1, ..., t_n -> t] is a subtype of the type [s_1, ..., s_m -> s] iff s is a subtype of t, $n = m$, and $s_i = t_i$ for $1 \leq i \leq n$.[13] This leads to subtype TCCs that prove 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: int): p(x1) IFF q(x1))
eq2_TCC1: OBLIGATION (FORALL (x1: int): p(x1) IFF TRUE)
```

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: `proj_1`, ..., `proj_`n, where each `proj_`i 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 for more details.

Record Types

Record types are of the form [# a_1:t_1, ..., a_n:t_n #]. The a_i are called *record accessors* and the t_i are types. Record types are similar to tuple types, except that the order is unimportant and there are named accessors for records. Associated with each record accessor a_i is a function of the same name of type [[# a_1:t_1,..., a_n:t_n #] -> t_i]. Record access in PVS utilizes these functions instead of the more usual

[12] These are defined in the prelude—see Appendix B.

[13] The alternative is to allow contravariant subtyping on the domains, *e.g.*, `[int -> int]` is a subtype of `[nat -> int]`.

"dot" notation, *e.g.*, $\mathtt{a}_i$`(r)` instead of `r.`$\mathtt{a}_i$. Record types are empty if any of the component types is empty.

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 ids need be associated only with those types on which later types depend.[14]

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 (LAMBDA (i:int), (j:int | i >= j): i-j)
```

[14]Ids may be provided for all of the types, but it tends to make the type specification cluttered and it becomes more difficult to see the dependencies.

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 9.

The language has a number of predefined operators. These are given in Figure 10 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 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 for details.

The infix *operator symbols* are `IFF`, `<=>`, `IMPLIES`, `WHEN`, `=>`, `OR`, `AND`, `&`, `=`, `/=`, `<`, `<=`, `>`, `>=`, `+`, `-`, `*`, `/` and `o`. Prefix operator symbols are `NOT` and `-`. These may all 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 equivalent in every respect. 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.

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 (see Section B). All of these have their standard meaning, except for `WHEN`, which is a reversed `IMPLIES` (*i.e.*, A `WHEN` $B \equiv B$ `IMPLIES` A).

Equality (`=`) and inequality (`/=`) 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 equality is of that type. 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}
```

$$
\begin{array}{lcl}
\mathit{expr} & \Rightarrow & \mathit{number} \\
 & | & \mathit{name} \\
 & | & \mathit{expr\ binop\ expr} \\
 & | & \mathit{unaryop\ expr} \\
 & | & \texttt{(}\ \mathit{expr}\ \texttt{)} \\
 & | & \mathit{tuple\text{-}expr} \\
 & | & \mathit{record\text{-}expr} \\
 & | & \mathit{set\text{-}expr} \\
 & | & \mathit{expr}\ :\ \mathit{type\text{-}expr} \\
 & | & \mathit{if\text{-}expr} \\
 & | & \mathit{expr\ arguments} \\
 & | & \mathit{bind\text{-}expr} \\
 & | & \mathit{update\text{-}expr} \\
 & | & \mathit{let\text{-}expr} \\
 & | & \mathit{where\text{-}expr} \\
 & | & \mathit{cases\text{-}expr}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{tuple\text{-}expr} & \Rightarrow & \texttt{(}\ \langle\, \mathit{expr}\, \rangle^{+}_{,}\ \texttt{)} \\
\mathit{record\text{-}expr} & \Rightarrow & \texttt{(\#}\ \langle\, \mathit{id}\ \texttt{:=}\ \mathit{expr}\, \rangle^{+}_{,}\ \texttt{\#)} \\
\mathit{set\text{-}expr} & \Rightarrow & \{\ \mathit{id}\ [\ :\ \mathit{type\text{-}expr}]\ \texttt{|}\ \mathit{expr}\ \} \\
\mathit{if\text{-}expr} & \Rightarrow & \texttt{IF}\ \mathit{expr}\ \texttt{THEN}\ \mathit{expr}\ \langle \texttt{ELSIF}\ \mathit{expr}\ \texttt{THEN}\ \mathit{expr} \rangle^{*}\ \texttt{ELSE}\ \mathit{expr}\ \texttt{ENDIF} \\
\mathit{bind\text{-}expr} & \Rightarrow & \mathit{bindop\ formals}\ [\texttt{->}\ \mathit{type\text{-}expr}]\ :\ \mathit{expr} \\
\mathit{bindop} & \Rightarrow & \texttt{LAMBDA} \mid \texttt{LAM} \mid \texttt{FORALL} \mid \texttt{ALL} \mid \texttt{EXISTS} \mid \texttt{SOME} \\
\mathit{update\text{-}expr} & \Rightarrow & \mathit{expr}\ \texttt{WITH [}\ \langle\, \mathit{assignment}\, \rangle^{+}_{,}\ \texttt{]} \\
\mathit{let\text{-}expr} & \Rightarrow & \langle \texttt{LET} \mid \texttt{PLET} \rangle\ \langle\, \mathit{expr}\ \texttt{=}\ \mathit{expr}\, \rangle^{+}_{,}\ \texttt{IN}\ \mathit{expr} \\
\mathit{where\text{-}expr} & \Rightarrow & \mathit{expr}\ \texttt{WHERE}\ \langle\, \mathit{expr}\ \texttt{=}\ \mathit{expr}\, \rangle^{+}_{,} \\
\mathit{cases\text{-}expr} & \Rightarrow & \texttt{CASES}\ \mathit{expr}\ \langle\, \mathit{expr}\ \texttt{:}\mathit{expr}\, \rangle^{*}\ [\texttt{ELSE}\ \mathit{expr}]\ \texttt{ENDCASES} \\
\mathit{assignment} & \Rightarrow & \langle\, \mathit{name} \mid \mathit{number} \mid \mathit{assign\text{-}args} \rangle\ \texttt{:=}\ \mathit{expr} \\
\mathit{assign\text{-}args} & \Rightarrow & \langle\, \mathit{arguments}\, \rangle^{*}_{,} \\
\mathit{arguments} & \Rightarrow & \texttt{(}\ \langle\, \mathit{expr}\, \rangle^{*}_{,}\ \texttt{)}
\end{array}
$$

Figure 9: Expression syntax

```
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`.

Operators	Associativity
FORALL, EXISTS, LAMBDA, IN	None
IFF, <=>	Right
IMPLIES, =>, WHEN	Right
OR, \/	Right
AND, &, /\	Right
NOT, ~	None
=, /=, ==, <, <=, >, >=	Left
WITH	Left
WHERE	Left
+, -	Left
, /	Left
-	None
o	Left
:	Left
[], <>	None
^	Left

Figure 10: Precedence Table

When the (in)equality is between terms of type `bool`, the semantics is the same as for `IFF`. There is a pragmatic difference in the way the PVS prover processes these operators. (In)equalities may be used for rewriting, which makes for efficient proofs but is incomplete, *i.e.*, the prover may fail to find the proof of a true formula. On the other hand the `IFF` form is complete, but may lead to a large number of cases. When in doubt, use the equality, as the prover provides commands which turn an equality into an `IFF`.

`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.

When the `ELSIF` clause is present, it is translated into a nested `IF-THEN-ELSE`. 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
```

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 `naturalnumber`. The relational operators on numeric types are `<`, `<=`, `>`, and `>=`. The numeric operators and axioms are in the prelude—see Section B. As with the boolean operators, all of these operators may be defined on new types and retain their infix precedences.

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 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 for details.

Binding Expressions

The binding expressions are those which create a local scope for variables, including the quantified expressions and λ-expressions. Binding expressions consist of an operator, a list of bindings, and an expression. The operator is one of the keywords `FORALL`, `ALL`, `EXISTS`, `SOME`, `LAMBDA`, or `LAM`.[15] 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:

[15] Set expressions are also binding expressions; see Section 5.

```
x,y,z,d,e: VAR num
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 `num`. 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`, which may be abbreviated to `ALL` and `SOME`, respectively. 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]`. `LAMBDA` may be abbreviated to `LAM`. In addition, when the range is `bool`, a lambda expression may be represented as a set expression; see Section 5.

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.[16]

`LET` and `WHERE` Expressions

`LET` and `WHERE` expressions are provided for convenience, providing alternative forms of expression. The `LET` expression has two forms, depending on whether the bindings are to be processed sequentially (`LET`) or in parallel (`PLET`). Here are some examples

```
x : int = 3
LET x:int = 2, y:int = x * x in x + y
PLET x:int = 2, y:int = x * x in x + y
x + y WHERE x:int = 2, y:int = x * x
```

The value of the `LET` and `WHERE` expressions is 6, while the value of the `PLET` expression is 11. These expression are internally translated to applications of lambda expressions, where the `LET` and `WHERE` expressions translate to

```
(LAMBDA (x:int) : (LAMBDA (y:int) : x + y)(x * x))(2)
```

and the `PLET` expression translates to

```
(LAMBDA (x:int), (y:int) : x + y)(2, x * x)
```

[16]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.

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

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.[17] 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 equal expressions.[18] Note that the latter form may also represent a type—this usually causes no problems as the context generally makes it clear which is expected. Section B provides the usual functions and properties of sets.

Tuple Expressions

A tuple expression of the type $[t_1, \ldots, t_n]$ has the form $(e_1, \ldots, e_n)$. 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 (section B) may be instantiated with `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]`.

Record Expressions

Given a record type $[\# \; a_1: t_1, \ldots, a_n: t_n \; \#]$, the a_i are referred to as *accessors*. The i^{th} component of an element `r` of this type is given by $a_i(r)$ (instead of the more usual $r.a_i$ notation).

Record expressions of this type are of the form $(\# \; a_1 := e_1, \ldots, a_n := e_n \; \#)$, where each e_i is of type t_i. The symbol `|->` may be used in place of `:=`. In typechecking record expressions, the system first collects all visible record types, and then chooses those that have the same fields as the record expression. This means that record expressions may not be given prior to the record type being specified, though it is not necessary to specify the record type at the top level. We chose this approach because otherwise dependent types lead to obscure error messages and TCCs. Note that 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

[17] The prelude theory `defined_types` also defines `PRED`, `predicate`, `PREDICATE`, and `SETOF` as alternate equivalents with different spellings.

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

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

leads to the (unprovable) TCC `4 < 3`.

Override Expressions

Functions and records may be "modified" by means of the override expression. The result of an override expression is a function 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))
```

As with record expressions, `|->` may be used in place of `:=`.[19]

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`. The domain of the override expression may be an extension of the domain of the given function, 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.

More complex overrides can be made; for example,

```
R: TYPE = [# a: int, b: [int -> [int,int -> int]] #]
r1: R
r2: r1 WITH [a |-> 0, (b)(1)(2,3) |-> 4]
```

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 used. For example, in

[19] We are exploring the possibility of separating the override expression into two parts: a *maplet*, which is the form following the with, and a `WITH` expression, that allows the override of one function with another. If this change takes place, it will have no effect on existing specifications.

```
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.

6 Theories

Specifications in PVS are built from *theories*, which provide genericity, reusability, and structuring. PVS theories may be parameterized, allowing for reuse. A theory consists of an *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 11.

specification	⇒	⟨ *theory* | *datatype* ⟩$^+$

theory	⇒	*id* [*theory-formals*] `: THEORY`
		  [*exportings*]
		`BEGIN`
		  [*assumings*]
		  [*theory*]
		`END` *id*

theory-formals	⇒	`[` ⟨ *theory-formal* ⟩$^+_,$ `]`
theory-formal	⇒	[`(` *importings* `)`] *theory-formal-decl*
theory-formal-decl	⇒	*theory-decl* | *simple-type-decl* | *simple-const-decl*

Figure 11: 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 `EXPORTING`s and `IMPORTING`s.

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`[20] 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 `ASSUMPTION`s. The body of the `groups` theory consists of two theorems, which can be proved from the assumptions.

[20] Recall that `o` is an infix operator.

```
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 12: Theory `groups`

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 prelude is detailed in Appendix B. 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.

Theory Parameters

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

Theory parameters may be types, subtypes, constants, or `IMPORTING`s.[21] 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). A theory is *instantiated* from within another theory

[21]There are plans to allow theories as parameters, along the lines of OBJ [6].

by providing *actual parameters* to substitute for the formals. Actual parameters may occur in IMPORTINGs, EXPORTINGs, theory declarations, and names. In each case they are enclosed in braces ([and]) and separated with commas.

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

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

a valid instance has four actual parameters; an example is

```
T[int, x:nat | x < 10, naturalnumbers.<=, 5, 6]
```

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

IMPORTINGs and EXPORTINGs

The IMPORTING and EXPORTING clauses form a horizontal hierarchy, much like the subroutine hierarchy of a programming language.[22]

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.

The EXPORTING clause

The EXPORTING clause specifies the names declared in the theory which are to be made available to the rest of the context. It may also specify instances of the theories which it used to be exported. The syntax of the EXPORTING clause is

exportings	$\Rightarrow$	EXPORTING *exporting-names* [WITH *exporting-theories*]
exporting-names	$\Rightarrow$	$\langle$ *expname* $\rangle^{+}_{,}$ | $\langle$ ALL [BUT $\langle$ *expname* $\rangle^{+}_{,}$] $\rangle$
expname	$\Rightarrow$	*idop* [: $\langle$ *type-expr* | TYPE | FORMULA $\rangle$]
exporting-theories	$\Rightarrow$	ALL | CLOSURE | $\langle$ *theoryname* $\rangle^{+}_{,}$

Figure 13: Exporting Syntax

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

[22] In the future, PVS will provide a means of expressing refinement, in which one theory *implements* another.

```
EXPORTING ALL WITH CLOSURE
```

Any declared name may be exported except for variable declarations. When `ALL` is specified for the *exporting-names*, 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`, 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) 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`s.

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.[23] 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.

`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

[23] Proofs are not used in completion analysis.

a theory. In addition, theory abbreviations implicitly use the theory name that they abbreviate (see Section 6).

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.[24]

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, so that a theory may not end up importing itself, directly or indirectly.

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.

Assuming Part

The assuming part consists of top-level declarations and `IMPORTING`s which are ordered. 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 14.

[24]Theory instances currently must be fully instantiated; in the future we may allow partially instantiated `IMPORTING`s.

assumings	⇒	ASSUMING ⟨ *assuming-decl* ⟩$^+$ ENDASSUMING
assuming-decl	⇒	*importings* | *mod-decl* | *type-decl* | *var-decl* | *const-decl* | *assumption*

Figure 14: Assuming Syntax

The primary purpose of the assuming part is to provide constraints on the use of the theory, by means of ASSUMPTIONs. 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 parameters, and when instantiated they become OBLIGATIONs 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.

A special case of an assuming occurs when the theory declares a formal type parameter. As described in Section 4, types must be shown to be nonempty if there is a constant declared involving that type. Thus if any of the constants declared in the theory involve the type, then a nonempty assuming is generated by the system, and will become an obligation for any type provided as an actual that is not already known to be nonempty. On the other hand, if there are formal constant parameters which involve the formal type, then no such assuming is generated, since the user of the theory will already be obligated to show that the type is nonempty.

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 15.

theory-part	$\Rightarrow$	$\langle$ *theory-decl* $\rangle^{+}$
theory-decl	$\Rightarrow$	*importings* | *theory-abbr* | *type-decl* | *var-decl* | *const-decl* | *def-decl* | *formula*

Figure 15: Theory Part Syntax

7 Name Resolution

Names in PVS are used to denote theories, variables, constants, and formulas. Names are introduced by declaration. The syntax of names is given in Figure 16.

<table>
<tr><td>module-name</td><td>⇒</td><td>name</td></tr>
<tr><td>idop</td><td>⇒</td><td>id | opsym</td></tr>
<tr><td>module-actuals</td><td>⇒</td><td><code>[</code> $\langle$ type-expr | expr $\rangle^{*}_{,}$ <code>]</code></td></tr>
<tr><td>name</td><td>⇒</td><td>idop [module-actuals] [<code>.</code> idop]</td></tr>
<tr><td>opsym</td><td>⇒</td><td>$\langle$ unaryop | binop $\rangle$</td></tr>
<tr><td>unaryop</td><td>⇒</td><td><code>-</code> | <code>NOT</code> | <code>~</code> | <code>[]</code> | <code><></code></td></tr>
<tr><td>binop</td><td>⇒</td><td><code>+</code> | <code>-</code> | <code>*</code> | <code>o</code> | <code>/</code> | <code>=</code> | <code>/=</code> | <code><</code> | <code><=</code> | <code>></code> | <code>>=</code></td></tr>
<tr><td></td><td>|</td><td><code>&</code> | <code>=></code> | <code><=></code> | <code>IFF</code> | <code>IMPLIES</code> | <code>OR</code> | <code>AND</code></td></tr>
<tr><td></td><td>|</td><td><code>\/</code> | <code>/\</code> | <code>==</code> | <code>^</code></td></tr>
<tr><td>formula-name</td><td>⇒</td><td><code>AXIOM</code> | <code>CLAIM</code> | <code>CONJECTURE</code> | <code>COROLLARY</code> | <code>FACT</code></td></tr>
<tr><td></td><td>|</td><td><code>FORMULA</code> | <code>LAW</code> | <code>LEMMA</code> | <code>OBLIGATION</code> | <code>POSTULATE</code></td></tr>
<tr><td></td><td>|</td><td><code>PROPOSITION</code> | <code>SUBLEMMA</code> | <code>THEOREM</code></td></tr>
</table>

Figure 16: Name Syntax

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 that `true < 3` contains a reference to this declaration, whereas `2 < 3` does not. If the use of the name is not enough to distinguish, coercion may be used to specify the signature directly (see Section 5). 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 *thid*, 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 uses *thid*:

- *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. 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 error than those further up. Note that even the top form may be ambiguous, as *id* may be declared more than once in *thid*. If this is the case, then either the context will disambiguate the name or the user will have to supply the type in the form of a coercion expression, *e.g.*, id: `nat`.

The actual rules governing the resolution of names will be given in the PVS semantics document, which has yet to be written. In general, the system does the intuitively obvious thing, and the error messages are reasonably clear about how ambiguities arise.

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* [*theory-formals*] `:DATATYPE` `BEGIN` [*assumings*] ⟨ *constructor-spec* `:`*id* ⟩ + `END` *id*
constructor-spec	⇒	*id* [`(` ⟨ *id* `:`*type-expr* ⟩$^{+}_{,}$ `)`]

Figure 17: Datatype Syntax

The syntax for PVS datatypes is given in Figure 17. Datatypes may appear at the *top-level* as with theory declarations, or *in-line* as a declaration within a theory.[25] Typechecking a top-level datatype named `foo` causes the generation of a new PVS file named `foo_adt.pvs` containing two theories as described below. Typechecking an in-line datatype has the effect of adding new declarations to the current theory, effectively replacing the in-line datatype. In-line datatypes are more restricted: they may not have formal parameters or assuming parts, and they will not generate the recursive combinator (described below). The declarations generated for an in-line datatype may be viewed using the PVS `M-x prettyprint-expanded` command (see [9]).

A datatype declaration of the form

```
adt: DATATYPE
  BEGIN
   cons1(acc11: T11, ..., acc1n1: T1n1): rec1
   ⋮
   consm(accm1: Tm1, ..., acc1nm: T1nm): recm
  END adt
```

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 on these constructor specifications:

- No single identifier may be used for two different constructors.

[25] Enumeration types are actually in-line datatypes—see Section 3.

- No single identifier may be used for two different recognizers.
- No identifier may be used as both a constructor and a recognizer.
- The datatype identifier may not be used as a recognizer or an accessor.
- Duplicate accessor identifiers are not allowed for a single constructor specification.

Datatypes are usually recursive; this is the case when the types of the accessors reference the datatype. There are restrictions on the use of the datatype in the type of an accessor:

- The only allowable types for accessors involving the datatype `adt` are:
  - `adt`
  - `list[adt]`
  - `sequence[adt]`
- There must be at least one constructor specification that has no recursive accessor types. This ensures that the datatype is well-defined.

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

- The datatype identifier is used to declare an uninterpreted type.
- Each recognizer is used to declare an uninterpreted subtype of the datatype.
- Each constructor and accessor is used to generate a constant declaration.
- There is an extensionality axiom generated for each constructor specification.
- There is an eta axiom generated for each constructor specification which 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.*,

  $\mathtt{acc}_{ij}\mathtt{(cons}_i\mathtt{(e}_{i1}\mathtt{,\ldots,\ e}_{im_i}\mathtt{)\ =\ e}_{ij}$
- A disjointness axiom is generated that says that no element of the datatype can belong to more than one recognizer subtype.
- An inclusive axiom is generated that says that every element of the datatype belongs to at least one recognizer subtype.
- An induction scheme is provided for proving properties of the datatype.

- Recursive combinators are generated for "mapping" over elements of the datatype.

These axioms are illustrated below for `stacks`. Note that for accessors whose type references the datatype, the induction schema and recursive combinators are handled specially.

Example: stacks

The `stacks` theory was specified in Section 1, and it was necessary to include some axioms to describe the desired behavior. An alternative specification for stacks is shown in Figure 18.

```
stack [t: TYPE]:  DATATYPE
 BEGIN
  empty: emptystack?
  push(top: t, pop: stack) : nonemptystack?
 END stack
```

Figure 18: Datatype `stack`

In this specification, `empty` and `push` are *constructors*, `top` and `pop` are *accessors*, and `emptystack?` and `nonemptystack?` are *recognizers* of the parameterized `stack` type.

The recursive combinator allows the specification of functions such as `length`[26]:

```
length((s:stack)): nat =
  stack_nat_rec(0, (LAMBDA (x:t),(n:nat): n + 1))
```

In addition to the above, there is a theory generated with the primitive recursion functional corresponding to the datatype with return type `range`. It takes one argument per constructor. This argument is of type `range` if the corresponding constructor has no arguments. If there are accessors, then this argument is a function with an argument type `t` for each non-recursive accessor field, and argument type `range` if the recursive type is of the form `s` where `s` is the datatype being defined, and of type `list[range]` for argument type `list[s]`, and of type `sequence[range]` for argument of type `sequence[s]`. For the `stack` theory the generated theory is given in Figure 21

When the accessor types involve the `list` or `sequence` types, the induction scheme and recursive combinators become more involved. Figure 22 is a contrived example, and the generated induction schema and recursive combinators are shown in Figure 23.

[26] In order to preserve soundness, PVS requires all user-defined recursive functions to include a *measure*, which is used to generate termination conditions. The primary use of the recursive combinator is to build measure functions for recursive definitions.

`CASES` Expressions

The `CASES` expression uses 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

```
stack_adt[T: TYPE]: THEORY
  BEGIN

  stack: TYPE

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

  empty: (empty?)

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

  top: [(nonempty?) -> T]

  pop: [(nonempty?) -> stack]

  empty_extensionality: AXIOM
    (FORALL (empty?_var: (empty?)), (empty?_var2: (empty?)):
       empty?_var = empty?_var2)

  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)

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

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

  pop_push: AXIOM
    (FORALL (push1_var: T), (push2_var: stack):
       pop(push(push1_var, push2_var)) = push2_var)

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

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

Figure 19: Theory `stack_adt` (continues)

```
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)))

stack_nat_rec((empty?_fun: nat), (nonempty?_fun: [T, nat -> nat])):
  [stack -> nat] =
      LAMBDA (stack_var: stack):
        CASES stack_var OF
          empty: empty?_fun,
          push(push1_var, push2_var):
              nonempty?_fun(push1_var,
                            stack_nat_rec(empty?_fun,
                                          nonempty?_fun)(push2_var))
        ENDCASES

stack_ordinal_rec((empty?_fun: ordinal),
                  (nonempty?_fun: [T, ordinal -> ordinal])):
  [stack -> ordinal] =
      LAMBDA (stack_var: stack):
        CASES stack_var OF
          empty: empty?_fun,
          push(push1_var, push2_var):
              nonempty?_fun(push1_var,
                            stack_ordinal_rec(empty?_fun,
                                              nonempty?_fun)(push2_var))
        ENDCASES

END stack_adt
```

Figure 20: Theory `stack_adt`

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, in the built-in list datatype

```
list[t:TYPE] : DATATYPE
  BEGIN
  null: null?
  cons (car: t, cdr :list) :cons?
  END list
```

the cases expression

```
  CASES x OF
    null : FALSE
    cons(y, z) : even?(y) AND null?(z)
```

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

  IMPORTING stack_adt[T]
  stack_rec((empty?_fun: range), (nonempty?_fun: [T, range -> range])):
    [stack -> range] =
        LAMBDA (stack_var: stack):
          CASES stack_var OF
            empty: empty?_fun,
            push(push1_var, push2_var):
                nonempty?_fun(push1_var,
                              stack_rec(empty?_fun,
                                        nonempty?_fun)(push2_var))
          ENDCASES

  END stack_rec_mod
```

Figure 21: Theory `stack_rec_mod`

```
adt: DATATYPE

BEGIN

bot: bottom?
con(i:int,
    simp:adt,
    sset:setof[adt],
    slist:list[adt],
    sseq:sequence[adt]) : con?

END adt
```

Figure 22: Datatype `adt`

```
  ENDCASES
```

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

```
  IF null?(x)
     THEN FALSE
     ELSE (LAMBDA (y:int),(z:list):
               even?(y) AND null?(z))(car(x), cdr(x))
  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 types. For example, if the `x` above is declared to be a subtype of `list` in which `null` is excluded, then the `null?` case can safely be left out, and a TCC will be generated that obligates the user to prove that

```
adt_adt: THEORY
  BEGIN
  ⋮
  adt_induction: AXIOM
    (FORALL (p: [adt -> boolean]):
       p(bot)
         AND
         (FORALL (con1_var: int), (con2_var: adt), (con3_var: setof[adt]),
                 (con4_var: list[adt]), (con5_var: sequence[adt]):
            p(con2_var)
              AND (FORALL (x: (con3_var)): p(x))
                AND every(p, con4_var) AND every(p, con5_var)
              IMPLIES
              p(con(con1_var, con2_var, con3_var, con4_var, con5_var)))
         IMPLIES (FORALL (adt_var: adt): p(adt_var)))

  adt_nat_rec((bottom?_fun: nat),
              (con?_fun: [int, nat, setof[nat], list[nat], sequence[nat] ->
                           nat])):
    [adt -> nat] =
        LAMBDA (adt_var: adt):
          CASES adt_var OF
            bot: bottom?_fun,
            con(con1_var, con2_var, con3_var, con4_var, con5_var):
                con?_fun(con1_var,
                         adt_nat_rec(bottom?_fun, con?_fun)(con2_var),
                         image(adt_nat_rec(bottom?_fun, con?_fun), con3_var),
                         map(adt_nat_rec(bottom?_fun, con?_fun), con4_var),
                         map(adt_nat_rec(bottom?_fun, con?_fun), con5_var))
          ENDCASES
  ⋮

  END adt_adt
```

Figure 23: Induction schema and recursive combinator for datatype `adt`

`x` is not `null`. There is a trade-off here between simpler specifications and simpler verifications; if the `null` 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 `null` 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.

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 `UPPERCASE`.
- 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 angled brackets, as in

  var-decl ⇒ *id-or-opsyms* : ⟨ `VARIABLE` | `VAR` ⟩ *type-expr*

- An optional part A of a clause is enclosed in square brackets: $[\,A\,]$.
- Iteration of a clause B one or more times is indicated by enclosing it in angled brackets followed by a plus sign: $\langle\, B \,\rangle^{+}$; repetition zero or more times is indicated by an asterisk instead of the plus sign: $\langle\, B \,\rangle^{*}$.
- A symbol below the plus or asterisk indicates a clause separator; for example, $\langle\, B \,\rangle^{*}_{,}$ indicates zero or more repetitions of the clause B separated by commas.
- All other characters appearing on the right-hand side of a syntax rule are literal lexical elements of the language.

Specification

specification	⇒	⟨ *theory* | *datatype* ⟩$^{+}$

theory	⇒	*id* [*theory-formals*] `: THEORY`
		  [*exportings*]
		`BEGIN`
		  [*assumings*]
		  [*theory*]
		`END` *id*

theory-formals	⇒	`[` ⟨ *theory-formal* ⟩$^{+}_{,}$ `]`
theory-formal	⇒	[`(` *importings* `)`] *theory-formal-decl*
theory-formal-decl	⇒	*theory-decl* | *simple-type-decl* | *simple-const-decl*

Exportings

exportings	$\Rightarrow$	`EXPORTING` *exporting-names* [`WITH` *exporting-theories*]
exporting-names	$\Rightarrow$	$\langle$ *expname* $\rangle^{+}_{,}$ | $\langle$ `ALL` [`BUT` $\langle$ *expname* $\rangle^{+}_{,}$] $\rangle$
expname	$\Rightarrow$	*idop* [`:` $\langle$ *type-expr* | `TYPE` | `FORMULA` $\rangle$]
exporting-theories	$\Rightarrow$	`ALL` | `CLOSURE` | $\langle$ *theoryname* $\rangle^{+}_{,}$

Assumings

assumings	$\Rightarrow$	`ASSUMING` $\langle$ *assuming-decl* $\rangle^{+}$ `ENDASSUMING`
assuming-decl	$\Rightarrow$	*importings* | *mod-decl* | *type-decl* | *var-decl* | *const-decl* | *assumption*

Theory Part

theory-part	$\Rightarrow$	$\langle$ *theory-decl* $\rangle^{+}$
theory-decl	$\Rightarrow$	*importings* | *theory-abbr* | *type-decl* | *var-decl* | *const-decl* | *def-decl* | *formula*

Declarations

simple-type-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^+_,$ `:` `TYPE` [`FROM` *type-expr*]
simple-const-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^+_,$ `:` *type-expr*
theory-abbr	$\Rightarrow$	*id* `:` `THEORY` `=` *theory-name*
type-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^+_,$ `:` `TYPE` [*type-def*]
type-def	$\Rightarrow$	`=` *enum-type* $\mid$ `=` *type-expr* $\mid$ `FROM` *type-expr*
var-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^+_,$ `:` `VAR` *type-expr*
const-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^+_,$ [*formals*] `:` *type-expr* [*const-value*]
def-decl	$\Rightarrow$	$\langle$ *idop* $\rangle^+_,$ [*formals*] `:` `RECURSIVE` *type-expr* `=` *expr* `MEASURE` *expr*
assumption	$\Rightarrow$	*id* `:` `ASSUMPTION` *formula-def*
formula	$\Rightarrow$	*id* `:` *formula-name* *formula-def*
ids-spec	$\Rightarrow$	$\langle$ *idop* $\rangle^+_,$ $\mid$ $\langle$ *idop* *formals* $\rangle$
formals	$\Rightarrow$	`(` $\langle$ *formal* $\rangle^+_,$ `)`
formal	$\Rightarrow$	$\langle$ *idop* $\rangle^+_,$ [`:` *type-expr*]
	$\mid$	`(` $\langle$ *idop* $\rangle^+_,$ `:` *type-expr* `)`
	$\mid$	`(` *idop* [`:` *type-expr*] `|` *expr* `)`

Type Expressions

type-expr	$\Rightarrow$	*type-name* $\mid$ *subtype* $\mid$ *function-type*
	$\mid$	*tuple-type* $\mid$ *record-type*
type-name	$\Rightarrow$	*name*
subtype	$\Rightarrow$	*expr-as-type* $\mid$ *set-subtype*
expr-as-type	$\Rightarrow$	`(` *expr* `)`
set-subtype	$\Rightarrow$	`{` *expr* `:` *type-expr* `|` *expr* `}` [`CONTAINING` *expr*]
function-type	$\Rightarrow$	[$\langle$ `FUNCTION` $\mid$ `ARRAY` $\rangle$] `[` $\langle$ *type-expr* $\mid$ *id* `:` *type-expr* $\rangle^+_,$ `->` *type-expr* `]`
enumeration-type	$\Rightarrow$	`{` $\langle$ *id* $\rangle^+_,$ `}`
tuple-type	$\Rightarrow$	`[` $\langle$ *type-expr* $\mid$ *id* `:` *type-expr* $\rangle^+_,$ `]`
record-type	$\Rightarrow$	`[#` $\langle$ *field-decl* $\rangle^+_,$ `#]`
field-decl	$\Rightarrow$	$\langle$ *id* $\rangle^+_,$ `:` *type-expr*

Expressions

$$
\begin{array}{lcl}
\textit{expr} & \Rightarrow & \textit{number} \\
 & | & \textit{name} \\
 & | & \textit{expr binop expr} \\
 & | & \textit{unaryop expr} \\
 & | & \texttt{(}\ \textit{expr}\ \texttt{)} \\
 & | & \textit{tuple-expr} \\
 & | & \textit{record-expr} \\
 & | & \textit{set-expr} \\
 & | & \textit{expr}\ \texttt{:}\ \textit{type-expr} \\
 & | & \textit{if-expr} \\
 & | & \textit{expr arguments} \\
 & | & \textit{bind-expr} \\
 & | & \textit{update-expr} \\
 & | & \textit{let-expr} \\
 & | & \textit{where-expr} \\
 & | & \textit{cases-expr}
\end{array}
$$

$$
\begin{array}{lcl}
\textit{tuple-expr} & \Rightarrow & \texttt{(}\ \langle\, \textit{expr} \,\rangle^{+}_{,}\ \texttt{)} \\
\textit{record-expr} & \Rightarrow & \texttt{(\#}\ \langle\, \textit{id}\ \texttt{:=}\ \textit{expr} \,\rangle^{+}_{,}\ \texttt{\#)} \\
\textit{set-expr} & \Rightarrow & \{\ \textit{id}\ [\ \texttt{:}\ \textit{type-expr}]\ \texttt{|}\ \textit{expr}\ \} \\
\textit{if-expr} & \Rightarrow & \texttt{IF}\ \textit{expr}\ \texttt{THEN}\ \textit{expr}\ \langle \texttt{ELSIF}\ \textit{expr}\ \texttt{THEN}\ \textit{expr} \rangle^{*}\ \texttt{ELSE}\ \textit{expr}\ \texttt{ENDIF} \\
\textit{bind-expr} & \Rightarrow & \textit{bindop formals}\ [\texttt{->}\ \textit{type-expr}]\ \texttt{:}\ \textit{expr} \\
\textit{bindop} & \Rightarrow & \texttt{LAMBDA} \mid \texttt{LAM} \mid \texttt{FORALL} \mid \texttt{ALL} \mid \texttt{EXISTS} \mid \texttt{SOME} \\
\textit{update-expr} & \Rightarrow & \textit{expr}\ \texttt{WITH [}\ \langle\, \textit{assignment} \,\rangle^{+}_{,}\ \texttt{]} \\
\textit{let-expr} & \Rightarrow & \langle \texttt{LET} \mid \texttt{PLET} \rangle\ \langle\, \textit{expr}\ \texttt{=}\ \textit{expr} \,\rangle^{+}_{,}\ \texttt{IN}\ \textit{expr} \\
\textit{where-expr} & \Rightarrow & \textit{expr}\ \texttt{WHERE}\ \langle\, \textit{expr}\ \texttt{=}\ \textit{expr} \,\rangle^{+}_{,} \\
\textit{cases-expr} & \Rightarrow & \texttt{CASES}\ \textit{expr}\ \langle\, \textit{expr}\ \texttt{:}\textit{expr} \,\rangle^{*}\ [\texttt{ELSE}\ \textit{expr}]\,\texttt{ENDCASES} \\
\textit{assignment} & \Rightarrow & \langle\, \textit{name} \mid \textit{number} \mid \textit{assign-args} \rangle\ \texttt{:=}\ \textit{expr} \\
\textit{assign-args} & \Rightarrow & \langle\, \textit{arguments} \,\rangle^{*}_{,} \\
\textit{arguments} & \Rightarrow & \texttt{(}\ \langle\, \textit{expr} \,\rangle^{*}_{,}\ \texttt{)}
\end{array}
$$

Names

module-name	⇒	*name*
idop	⇒	*id* | *opsym*
module-actuals	⇒	`[` ⟨ *type-expr* | *expr* ⟩$^*_{,}$ `]`
name	⇒	*idop* [*module-actuals*] [`.` *idop*]
opsym	⇒	⟨ *unaryop* | *binop* ⟩
unaryop	⇒	`-` | `NOT` | `~` | `[]` | `<>`
binop	⇒	`+` | `-` | `*` | `o` | `/` | `=` | `/=` | `<` | `<=` | `>` | `>=`
	|	`&` | `=>` | `<=>` | `IFF` | `IMPLIES` | `OR` | `AND`
	|	`\/` | `/\` | `==` | `^`
formula-name	⇒	`AXIOM` | `CLAIM` | `CONJECTURE` | `COROLLARY` | `FACT`
	|	`FORMULA` | `LAW` | `LEMMA` | `OBLIGATION` | `POSTULATE`
	|	`PROPOSITION` | `SUBLEMMA` | `THEOREM`

Identifiers

id	⇒	*letter* ⟨ *ident-char* ⟩*
ident-char	⇒	*letter* | *digit* | `_` | `?`
letter	⇒	*A* | ... | *Z* | *a* | ... | *z*
digit	⇒	*0* | ... | *9*

Datatypes

datatype	⇒	*id* [*theory-formals*] `:DATATYPE` `BEGIN` [*assumings*] ⟨ *constructor-spec* `:` *id* ⟩$^+$ `END` *id*
constructor-spec	⇒	*id* [`(` ⟨ *id* `:` *type-expr* ⟩$^+_{,}$ `)`]

B The Prelude

Booleans

The `booleans` theory provides the declaration for the `boolean` type and operators. The semantics is given by the prover, which is classical. The `when` operator is a backwards `implies`, *e.g.*, `A WHEN B` is equivalent to `B IMPLIES A`.

```
% booleans declares the type boolean and its abbreviation bool, along
% with the usual boolean constants and connectives.  The prover provides
% the meaning for these connectives.

booleans: THEORY

 BEGIN

  boolean: TYPE
  bool: TYPE = boolean
  true, false: boolean
  not: [bool -> bool]
  and, &, or, implies, =>, when, iff, <=>: [bool,bool -> bool]

 END booleans
```

Equalities

Equality is declared in the `equalities` theory; like `booleans`, its semantics is defined by the prover. The theory does give the properties that make equality aan equivalence relation. Note that, because of the way it is declared, `A = B` is a legal expression only if a common type can be found for `A` and `B`.

```
% equalities contains the declaration for =, plus properties stating
% that it is an equivalence relation

equalities [t: TYPE] : THEORY

 BEGIN

  =: [t, t -> boolean]

  reflexivity_of_equals : POSTULATE (FORALL (x : t) : x = x)

  transitivity_of_equals : POSTULATE
      (FORALL (x, y, z: t) : x = y AND y = z
                     IMPLIES x = z)

  symmetry_of_equals : POSTULATE
      (FORALL (x, y : t) : x = y IMPLIES y = x)

 END equalities
```

```
% notequal defines /= in terms of =.  It is in a separate theory in
% order to properly bootstrap the system.

notequal [t: TYPE] : THEORY

 BEGIN

  /=: [t, t -> boolean] = (LAMBDA (a,b:t): NOT(a=b))

 END notequal
```

The `/=` operator is defined in a separate theory from `=` because the polymorphic `=` is needed to create axioms from definitions.[27]

if_def

The polymorphic `IF` is defined below. It is actually a function application, with the `IF-THEN-ELSE` simply an alternate form. The meaning of the `IF` operator is given by the three properties that follow.

[27] These are needed in order to support the prover.

```
% if_def provides the polymorphic declaration of the IF-THEN-ELSE
% connective, along with some basic properties.  Note that the
% declaration for IF is for a 3-ary function, and that the IF-THEN-ELSE
% form is simply an alternative form.

if_def [t: TYPE] : THEORY

 BEGIN

  IF:[bool,t,t -> t]

  IF_true : POSTULATE
    (FORALL (x, y : t) : IF true THEN x ELSE y ENDIF = x)

  IF_false : POSTULATE
    (FORALL (x, y : t) : IF false THEN x ELSE y ENDIF = y)

  IF_same : LEMMA
    (FORALL (b : bool), (x : t) : IF b THEN x ELSE x ENDIF = x)

 END if_def
```

defined_types

The theory defined_types defines the predicate and setof types.

```
% defined_types provides the declarations for types pred and setof

defined_types [t: TYPE] : THEORY

 BEGIN
  pred: TYPE = [t -> bool]
  PRED: TYPE = [t -> bool]
  predicate: TYPE = [t -> bool]
  PREDICATE: TYPE = [t -> bool]
  setof: TYPE = [t -> bool]
  SETOF: TYPE = [t -> bool]
 END defined_types
```

Epsilon

The epsilon theory defines the choice function epsilon, which corresponds to Hilbert's ε-operator. Given a predicate over a type t, epsilon chooses an element of t that satisfies the predicate, or an arbitrary element of t otherwise. This choice is always the same one for a given type and predicate, though if more than one element of the type satisfies the predicate, there is no way to tell which is chosen. The important property is epsilon_ax, which says that if anything satisfies the predicate, then the chosen element does.

```
% epsilons provides a "choice" function; given a predicate over a type
% t, epsilon produces an element of t, which satisfies the predicate if
% any element of t does - otherwise it is an arbitrary element of t.
% Note that the original type t must be nonempty.

epsilons [t : TYPE] : THEORY
 BEGIN

  epsilon: [pred[t] -> t]

  epsilon_ax: AXIOM
        FORALL (p: pred[t]) :
                (EXISTS (x: t) : p(x)) => p(epsilon(p))

 END epsilons
```

Functions

The theory `functions` defines `extensionality` and the `eta` formulas, which are used to show that two functions are equal. The `injective?` predicate is true for one-to-one functions, `surjective?` holds for onto functions, and `bijective?` is true when both propeerties are satisfied. `inverse` is defined in terms of the `epsilon` function, and is that function which, given an element `y` of the range chooses an element `x` of the domain such that the function applied to `x` yields `y`. This definition of inverse is defined for all functions, but only if the function is injective will it be possible to say much about the inverse. See the writeup on the proof of the Schröder-Bernstein theorem for more discussion.

```
% functions provides the basic properties of functions.  Because of the
% type equivalence of [[t1,...,tn] -> t] and [t1,...,tn -> t], this
% theory handles any function arity.  Note that the definition of
% inverse uses epsilon; in practice this definition will only be useful
% for injective functions.

functions [dom,rng: TYPE] : THEORY

 BEGIN

  extensionality : AXIOM
     (FORALL (f, g : [dom -> rng]) :
        (FORALL (x : dom) : f(x) = g(x) IMPLIES f = g))

  eta : LEMMA
    (FORALL (f : [dom -> rng]) :
       (LAMBDA (x : dom) : f(x)) = f)

  injective? : pred[[dom -> rng]]
        = (LAMBDA (f:[dom->rng]):
           (FORALL (x,y:dom): (f(x) = f(y) => (x = y))))

  surjective? : pred[[dom -> rng]]
        = (LAMBDA (f:[dom->rng]):
           (FORALL (y:rng): (EXISTS (x:dom): f(x) = y)))

  bijective? : pred[[dom -> rng]]
        = (LAMBDA (f:[dom->rng]): injective?(f) & surjective?(f))

  inverse : [[dom -> rng] -> [rng -> dom]]
        = (LAMBDA (f:[dom->rng]): (LAMBDA (y:rng):
                (epsilon(lambda (x:dom): f(x) = y))))

  bijective_inverse: LEMMA
    (FORALL (f:(bijective?)),(x:dom),(y:rng):
       inverse(f)(y) = x IFF y = f(x))

 END functions
```

Sets

The `sets` theory defines the polymorphic `set` type and the usual set operations. It is important to remember that sets are just predicates.

```
% sets provides the polymorphic set type, along with the usual set
% operations.  It is important to bear in mind that a set is equivalent
% to a predicate.

sets [T: TYPE] : THEORY

 BEGIN

  set: TYPE = setof[T]

  member(x:T,a:set): bool = a(x)

  union(a,b:set): set =
    {x:T | member(x,a) OR member(x, b)}

  intersection(a,b:set): set =
    {x:T | member(x,a) AND member(x, b)}

  difference(a,b:set) : set =
    {x:T | member(x,a) AND NOT member(x, b)}

  add(x:T,a:set) : set =
    {y:T | x = y OR member(y, a)}

  remove(x:T,a:set) : set =
    {y:T | x /= y AND member(y, a)}

  singleton(x:T) : set = {y:T | y = x}

  subset?(a,b:set) : bool = (FORALL (z:T) : member(z, a) => member(z, b))

  strict_subset?(a,b:set) : bool = subset?(a, b) & a /= b

  empty?(a:set) : bool = (FORALL (x:T) : NOT member(x,a))

  emptyset: set = {x:T | false}

  nonempty?(a:set) : bool = NOT empty?(a)

  fullset: set = {x:T | true}

  disjoint?(a,b:set) : bool = empty?(intersection(a,b))

  extensionality: LEMMA
    FORALL (a,b: set) :
      (FORALL (x:T) : member(x, a) = member(x, b)) => (a = b)

 END sets
```

Function Properties

Theory `identity` is just the polymorphic identity function. There are three names given to this function.

```
% identity simply defines the identity function.

identity[t:TYPE] : THEORY
 BEGIN
  I: (bijective?[t,t]) = (LAMBDA (x:t): x)
  id: (bijective?[t,t]) = I
  identity: (bijective?[t,t]) = I
 END identity
```

Theory `function_image` defines the `image` and `inverse_image` of a function over a set.

```
% function_image provides the image and inverse_image functions.
% inverse_image is not the same as inverse; it is defined for all
% functions, not just injections.

function_image [dom, rng : TYPE] : THEORY

 BEGIN

  image : [[dom -> rng], set[dom] -> set[rng]]
        = (LAMBDA (f:[dom->rng]),(X:set[dom]):
            {y:rng | (EXISTS (x:(X)): y = f(x))})

  inverse_image : [[dom -> rng], set[rng] -> set[dom]]
        = (LAMBDA (f:[dom->rng]),(Y:set[rng]):
            {x:dom | member(f(x), Y)})

END function_image
```

Theories `function_props` and `function_props2` define composition of functions and give the associativity of composition, respectively.

```
% functions_props defines function composition.  It can't be defined in
% functions because it involves three type parameters.

function_props [t1,t2,t3: TYPE] : THEORY
 BEGIN

  o(f:[t2->t3],g:[t1->t2]): [t1->t3] = (LAMBDA (x:t1): f(g(x)))

 END function_props
```

```
% function_props2 defines the associativity of function composition.  It
% can't be defined in function_props because it involves four type
% parameters.

function_props2 [t1,t2,t3,t4: TYPE] : THEORY
 BEGIN
  assoc: LEMMA (FORALL (f:[t3->t4]),(g:[t2->t3]),(h:[t1->t2]):
                (f o g) o h = f o (g o h))
 END function_props2
```

Numbers

PVS has all the usual numeric types defined in the prelude. Note that the construction is in the opposite direction from the usual construction given in mathematics, where natural numbers are given,[28] and the integers, rationals, and reals are constructed in turn. In PVS, the important property is that of being a subtype, and subtypes are constructed in the opposite order.

The type `number` is not actually defined, and may reflect the complex numbers, the extended reals, etc. It is provided because the union type is not yet available, so there is no convenient way to define a supertype of the reals. The usual operators are defined on this type, and their projections onto the reals should behave as expected, *e.g.*, when `a` and `b` are reals, `numbers.+(a,b)` is the sum of `a` and `b`.

[28] God made the natural numbers, everything else is the work of man—*Kronecker*.

```
% numbers is the most general of the numeric types.  It defines the
% usual operators.  Note that the type of the denominator is nonzero,
% and that '-' is overloaded.

numbers: THEORY

 BEGIN

  number: TYPE
  nonzero_number: TYPE = {r: number | r /= 0} CONTAINING 1;

  +: [number,number -> number]
  -: [number,number -> number]
  *: [number,number -> number]
  /: [number,nonzero_number -> number]

  -: [number -> number]

  <: [number,number -> bool]
  <=(x,y:number): bool = x < y OR x = y;
  >(x,y:number): bool = y < x;
  >=(x,y:number): bool = y <= x;

 END numbers
```

The real numbers are declared to be an uninterpreted subtype of `number`. These are the mathematical real numbers, not the floating- or fixed-point numbers of a programming language. The PVS system does not currently have support for directly specifying numeric real constants other than the integers; this is generally not a problem, since most constants should be specified abstractly anyway. For example, any decimal number for π would be an approximation at best; it is much better simply to declare a constant `pi` and define its properties.

```
% reals defines the real numbers as an uninterpreted subtype of number.
% The basic number operations are redeclared in order to specify that
% they are closed, e.g. the sum of two reals is a real number.

reals: THEORY

 BEGIN

  real: TYPE FROM number
  nonzero_real: TYPE = {r: real | r /= 0} CONTAINING 1;
  nzreal: TYPE = nonzero_real

  closed_plus: AXIOM (FORALL (x,y:real): real_pred(x + y))
  closed_minus: AXIOM (FORALL (x,y:real): real_pred(x - y))
  closed_times: AXIOM (FORALL (x,y:real): real_pred(x * y))
  closed_divides: AXIOM (FORALL (x:real),(y:nzreal): real_pred(x / y))
  closed_neg: AXIOM (FORALL (x:real): real_pred(-x)) ;
  +(x,y:real): real = numbers.+(x,y) ;
  -(x,y:real): real = numbers.-(x,y) ;
  *(x,y:real): real = numbers.*(x,y) ;
  /(x:real, y:nzreal): real = numbers./(x,y) ;
  -(x:real): real = numbers.-(x) ;

 END reals
```

The rational numbers are an uninterpreted subtype of the reals. The basic operators have been redefined in order to maintain closure, which simplifies typechecking. A rational number may be given as a division, *e.g.*, `22 / 7`.

```
% rationals defines the rational numbers as an uninterpreted subtype of real.
% The basic number operations are redeclared in order to specify that
% they are closed, e.g. the sum of two rationals is a rational.

rationals: THEORY

 BEGIN

  rational: TYPE FROM real
  rat: TYPE = rational

  nonzero_rational: TYPE = {r: rational | r /= 0} CONTAINING 1
  nzrat: TYPE = nonzero_rational

  closed_plus: AXIOM (FORALL (x,y:rat): rational_pred(x + y))
  closed_minus: AXIOM (FORALL (x,y:rat): rational_pred(x - y))
  closed_times: AXIOM (FORALL (x,y:rat): rational_pred(x * y))
  closed_divides: AXIOM (FORALL (x:rat),(y:nzrat): rational_pred(x / y))
  closed_neg: AXIOM (FORALL (x:rat): rational_pred(-x)) ;
  +(x,y:rat): rat = numbers.+(x,y);
  -(x,y:rat): rat = numbers.-(x,y);
  *(x,y:rat): rat = numbers.*(x,y);
  /(x:rat,y:nzrat): rat = numbers./(x,y);

  -(x:rat): rat = numbers.-(x);

 END rationals
```

The integers are an uninterpreted subtype of the rationals. Again, the basic operators are redefined to be closed on the integers.

```
% integers defines the integers as an uninterpreted subtype of rational.
% The basic number operations are redeclared in order to specify that
% they are closed, e.g. the sum of two integers is an integer.

integers: THEORY

 BEGIN

  integer: TYPE FROM rational
  int: TYPE = integer

  closed_plus: AXIOM (FORALL (x,y:int): integer_pred(x + y))
  closed_minus: AXIOM (FORALL (x,y:int): integer_pred(x - y))
  closed_times: AXIOM (FORALL (x,y:int): integer_pred(x * y))
  closed_neg: AXIOM (FORALL (x:int): integer_pred(-x)) ;
  +(x,y:int): int = numbers.+(x,y) ;
  -(x,y:int): int = numbers.-(x,y) ;
  *(x,y:int): int = numbers.*(x,y) ;
  -(x:int): int = numbers.-(x) ;

 END integers
```

The natural numbers are an interpreted subtype of the integers, consisting of those integers which are not less than zero. All numeric constants (0, 1, 2, ...) are natural numbers. Closely associated with the naturals are natural number induction and cardinality, described below.

```
% naturalnumbers defines the natural numbers as a subtype of integer.
% The sum and product operations are redeclared in order to specify that
% they are closed, e.g. the sum of two natural numbers is a natural
% number.  The posnat type is defined here, as are the successor and
% predecessor functions.

naturalnumbers: THEORY

 BEGIN

  naturalnumber: TYPE = {x:integer | x >= 0} CONTAINING 0
  nat : TYPE = naturalnumber
  posnat: TYPE = {n:nat | n > 0}
  posint: TYPE = posnat

  +(x,y:nat): nat = numbers.+(x,y) ;
  *(x,y:nat): nat = numbers.*(x,y) ;

  suc(x:nat): nat = x + 1

  prd(x:nat): nat = IF x > 0 THEN x - 1 ELSE 0 ENDIF

 END naturalnumbers
```

The `nat_types` theory provides the two most useful bounded subtypes of the naturals. These are frequently used to indicate a finite set of distinct elements, where it is not desired to pin down the actual number of elements. Associated with these types is bounded induction, described below.

```
% nat_types defines some useful subtypes of the natural numbers.

nat_types[m: nat] : THEORY
 BEGIN
  upto:   TYPE = {s: nat | s <= m} CONTAINING m
  upfrom: TYPE = {s: nat | s >= m} CONTAINING m
  below:  TYPE = {s: nat | s < m}
  above:  TYPE = {s: nat | s > m} CONTAINING m + 1
 END nat_types
```

Iteration

The `function_iterate` theory provides the `iterate` function, where `iterate(f,n)` is the composition of `f` with itself `n` times, normally written as $\mathtt{f}^{\mathtt{n}}$ in mathematical notation. `iterate(f,0)` is the identity, and `iterate(f,1)` is just `f`.

```
% function_iterate provides the iterate function, which composes a
% function with itself a specified number of times.  The function must
% have the same domain and range.  Two useful properties of iterate are
% also provided.

function_iterate[t: TYPE] : THEORY
 BEGIN

  iterate(f: [t -> t], n: nat): RECURSIVE [t -> t] =
    (LAMBDA (x: t):
       IF n = 0 THEN x ELSE iterate(f, n-1)(f(x)) ENDIF)
    MEASURE (LAMBDA (f: [t -> t]), (n: nat): n)

  iterate_add: LEMMA
    (FORALL (f: [t -> t]), (m, n: nat):
      iterate(f,m) o iterate(f,n) = iterate(f, m + n))

  iterate_mult: LEMMA
    (FORALL (f: [t -> t]), (m, n: nat):
      iterate(iterate(f,m), n) = iterate(f, m * n))

 END function_iterate
```

Sequences

Sequences in PVS are defined as functions whose domain is the natural numbers. Note that sequences are infinite, and finite sequences are not a subtype of sequences.

There are ways to represent finite sequences in terms of infinite sequences so that they are subtypes. For example, by choosing an element of the base type, and specifying that finite sequences are those that have some suffix all of whose elements are that element. Thus the empty sequence would be given by `(LAMBDA (x:t) a)`, where `a` is the chosen element.[29]

Sequences are zero-based, so the first element of a sequence `s` is `s(0)`. The `nth` operator extracts the specified element from a sequence. `suffix` constructs a new sequence by ignoring the first `n` elements of a given sequence. `first` is the first (*i.e.*, zeroth) element of the sequence, and `rest` is the sequence got by dropping the first element. `delete` constructs a new sequence in which the specified element is deleted, and insert inserts a new element into a sequence at a specified location, shifting the remaining eleents of the sequence. `add` simply adds a new element to the front of a sequence. `every` takes a predicate and a sequence and returns true if every element of the sequence satisfies the predicate. Finally, `seq_induction` provides an induction schema for sequences, based on natural number induction.

[29]Since this is just one possible way of handling finite sequences, and has some disadvantages, it is not included in the prelude.

```
% sequences provides the polymorphic sequence type, as a function from
% nat to the base type.  The usual sequence functions are also provided.
% Note that these are infinite sequences, and do not contain finite
% sequences as a subtype.

sequences[t: TYPE] : THEORY

 BEGIN

  sequence : TYPE = [nat->t]

  nth(seq: sequence, n: nat): t = seq(n)

  suffix(seq:sequence, n:nat): sequence =
    (LAMBDA (i:nat): seq(i+n))

  first(seq: sequence): t = nth(seq, 0)

  rest(seq: sequence): sequence = suffix(seq, 1)

  delete(n : nat, seq : sequence) : sequence =
      (LAMBDA (i : nat) :
             (IF i < n THEN seq(i)
                 ELSE seq(i + 1) ENDIF))

  insert(x:t, n:nat, seq:sequence) : sequence =
         (LAMBDA (i : nat) :
             (IF i < n THEN seq(i)
               ELSIF i = n THEN x
                 ELSE seq(i - 1) ENDIF))

  add(x:t, seq:sequence) : sequence =
     insert(x, 0, seq)

  insert_delete : LEMMA
    (FORALL (n : nat), (seq : sequence) :
      insert(nth(seq, n), n, delete(n, seq)) = seq)

  add_first_rest : LEMMA
    (FORALL (seq : sequence) :
      add(first(seq), rest(seq)) = seq)

  every(p: pred[t], seq: sequence): bool =
    (FORALL (n:nat): p(nth(seq, n)))

  seq_induction : LEMMA
    (FORALL (p: pred[t]), (seq: sequence) :
         p(nth(seq,0)) AND
         (FORALL (n : nat) : p(nth(seq, n)) IMPLIES p(nth(seq, n + 1)))
      IMPLIES every(p, seq))

 END sequences
```

Theory `seq_functions` provides the `map` function, which maps a function over a sequence, returning another sequence.

```
% seq_functions defines the map function that generates the image of a
% sequence under a function.

seq_functions[t1,t2: TYPE] : THEORY

 BEGIN
  map(f: [t1 -> t2], s: sequence[t1]): sequence[t2] =
    (LAMBDA (n: nat): f(nth(s,n)))

 END seq_functions
```

Relations

The `relations` theory defines various predicates on binary relations over a single type.

```
% relations defines properties that are useful in specifying binary
% relations.

relations [T : TYPE] : THEORY
 BEGIN

  reflexive?(R: pred[[T,T]]) : bool =
    FORALL (x:T) : R(x, x)

  irreflexive?(R: pred[[T,T]]) : bool =
    FORALL (x:T) : NOT R(x, x)

  symmetric?(R: pred[[T,T]]) : bool =
    FORALL (x, y:T) : R(x, y) IMPLIES R(y, x)

  antisymmetric?(R: pred[[T,T]]) : bool =
    FORALL (x,y:T) : R(x, y) & R(y, x) => x = y

  transitive?(R: pred[[T,T]]) : bool =
    FORALL (x,y,z:T) : (R(x, y) & R(y, z) => R(x, z))

END relations
```

The `relation_image` theory defines the image and converse of a relation.

```
% relation_image defines the image and the converse of a relation.

relation_image [dom, rng : TYPE] : THEORY
 BEGIN

 image(R : pred[[dom, rng]], D : set[dom]) : set[rng] =
   {y : rng | (EXISTS (x : (D)) : R(x, y))}

 converse(R : pred[[dom, rng]]) : pred[[rng, dom]] =
   (LAMBDA (y : rng), (x : dom) : R(x, y))

 isomorphism?(R : pred[[dom, rng]]) : bool =
   (EXISTS (f : (bijective?[dom, rng])):
     (FORALL (x: dom) :(FORALL (y:rng): R(x, y) IFF f(x) = y)))

END relation_image
```

Orders

Theory `orders` defines the ordering relations `pre_order?`, `partial_order?`, `strict_order?`, `total_order?`, `strict_total_order?`, and `well_founded?`.

```
% orders defines the usual ordering relations.

orders [T : TYPE] : THEORY
 BEGIN

  pre_order?(<=: pred[[T,T]]) : bool =
    reflexive?(<=) & transitive?(<=)

  partial_order?(<=: pred[[T,T]]) : bool =
   pre_order?(<=) &
   antisymmetric?(<=)

  strict_order?(<: pred[[T,T]]) : bool =
    irreflexive?(<) & antisymmetric?(<) & transitive?(<)

  total_order?(<=: pred[[T,T]]) : bool =
    partial_order?(<=) & FORALL (x,y:T) : x <= y OR y <= x

  tricotomous?(<: pred[[T,T]]) : bool =
    FORALL (x,y:T) : x < y OR y < x OR x = y

  strict_total_order?(<: pred[[T,T]]) : bool =
    strict_order?(<) & tricotomous?(<)

  well_founded?(<=: pred[[T,T]]): bool =
     (FORALL (p: pred[T]):
        (EXISTS (y:T): p(y))
           IMPLIES (EXISTS (y:(p)):
                      (FORALL (x:(p)): (x <= y IMPLIES x = y))))

 END orders
```

Induction

The following theories all define various forms of induction. Note that only `wf_induction` is an axiom, the rest follow from it.

```
% wf_induction defines induction for any type that has a well-founded
% relation.

wf_induction [T: type, <= : (well_founded?[T])] : THEORY

 BEGIN

  wf_induction: AXIOM
    (FORALL (p : pred[T]):
      (FORALL (x : T):
        (FORALL (y : T): y<=x AND y/= x IMPLIES p(y))
           IMPLIES p(x))
    IMPLIES
      (FORALL (x:T): p(x)))

 END wf_induction
```

```
% measure_induction builds on well-founded induction.  It allows
% induction over a type T for which a measure function m is defined.

measure_induction
  [T, M : TYPE, m : [T -> M], <= : (well_founded?[M])] :  THEORY

 BEGIN

 measure_induction : LEMMA
  (FORALL (p : pred[T]) :
     (FORALL (x : T) :
         (FORALL (y : T): m(y) <= m(x) AND m(y) /= m(x) IMPLIES  p(y))
             IMPLIES p(x))
    IMPLIES (FORALL (x : T) : p(x)))

 END measure_induction
```

```
% nat_induction provides both weak and strong induction on natural numbers.

nat_induction : THEORY
 BEGIN

  wf_nat: AXIOM well_founded?(LAMBDA (m,n:nat): m <= n)

  nat_induction: LEMMA
    (FORALL (p: pred[nat]) :
       (p(0) AND (FORALL (j: nat): p(j) IMPLIES p(j+1)))
           IMPLIES (FORALL (i: nat):p(i)))

  %Strong induction on naturals.
  NAT_induction : LEMMA
    (FORALL (p: pred[nat]) :
        (FORALL (j: nat): (FORALL (k : nat) : k < j IMPLIES p(k)) IMPLIES p(j))
           IMPLIES (FORALL (i: nat):p(i)))

 END nat_induction
```

```
% bounded induction provides induction over bounded natural numbers.

bounded_induction [m: nat] : THEORY
 BEGIN

  upto_induction: LEMMA
    (FORALL (p : pred[upto[m]]):
      (p(0) AND (FORALL (r: upto[m]): r < m AND p(r) IMPLIES p(r + 1)))
          IMPLIES (FORALL (s: upto[m]): p(s)))

  upfrom_induction: LEMMA
    (FORALL (p : pred[upfrom[m]]):
      (p(m) AND (FORALL (r: upfrom[m]): p(r) IMPLIES p(r + 1)))
           IMPLIES (FORALL (s: upfrom[m]): p(s)))

  below_induction: LEMMA
    (FORALL (p : pred[below[m]]):
      (m > 0 IMPLIES
         (p(0) AND (FORALL (r: below[m]): r < m - 1 AND p(r) IMPLIES p(r + 1)))
             IMPLIES (FORALL (s: below[m]): p(s))))

  above_induction: LEMMA
    (FORALL (p : pred[above[m]]):
      (p(m+1) AND (FORALL (r: above[m]): p(r) IMPLIES p(r + 1)))
          IMPLIES (FORALL (s: above[m]): p(s)))

 END bounded_induction
```

Cardinalities

There are three theories defining cardinality. All three of these theories are parameterized by a type, a function that maps the natural numbers to the type, and a bound. The most general theory is `cardinality`, which is valid for any type, but leads to complex TCCs with each use. If it is known that the type is finite, then `finite_cardinality` may be easier to work with, as the assumings need only be discharged once for the type, and no further TCCs are needed, since any subset of a finite type is finite. Between these two theories is `countable_cardinality`, which applies to countable types (usually the natural numbers). In this case there is are both assumings and TCCs to deal with, but the TCCs are simpler than those of `cardinality`.

```
% cardinality provides the card function, which gives the cardinality of
% a finite set.  The f parameter gives a mapping from the naturals to
% the base type of the sets, and the m parameter gives an upper bound on
% the cardinality.  This is the most general cardinality function,
% making no assumptions on the base type T.  If T is known to be
% countable or finite, then the corresponding cardinality theories are
% probably more appropriate.

cardinality[T: TYPE, m : nat, f : [below[m] -> T]] : THEORY
 BEGIN

  fin : TYPE =
    {s : set[T] | (FORALL (x: (s)): (EXISTS (i:below[m]): f(i) = x)) AND
                  (FORALL (i, j: below[m]): s(f(i)) AND
                                     f(i) = f(j) IMPLIES i = j)}
        CONTAINING emptyset[T]

  cardi(s: fin, i: below[m]) : RECURSIVE nat =
    (IF i = 0 THEN IF s(f(i)) THEN 1 ELSE 0 ENDIF
             ELSIF s(f(i)) THEN cardi(s, i-1) + 1
             ELSE cardi(s, i-1) ENDIF)
    MEASURE (LAMBDA (s: fin), (i: below[m]): i)

  card(s: fin) : nat = IF m = 0 THEN 0 ELSE cardi(s, m-1) ENDIF

 END cardinality
```

```
% countable_cardinality provides the ccard function, which gives the
% cardinality of a finite set whose base type is known to be countable.
% The f parameter gives a bijective mapping from the naturals to the base
% type of the sets, and the m parameter gives an upper bound on the
% cardinality.  This form of cardinality is useful when the base type is
% countable, as the finite subsets are easier to deal with.

countable_cardinality[T: TYPE, m: nat, f : (injective?[below[m], T])] : THEORY
 BEGIN

  cfin : TYPE =
    {s : set[T] | (FORALL (x: (s)) :(EXISTS (i: below[m]): f(i) = x))}
    CONTAINING emptyset[T]

  ccardi(s: cfin, i: below[m]) : RECURSIVE nat =
    (IF i = 0 THEN IF s(f(i)) THEN 1 ELSE 0 ENDIF
              ELSIF s(f(i)) THEN ccardi(s, i-1) + 1
              ELSE ccardi(s, i-1) ENDIF)
    MEASURE (LAMBDA (s: cfin), (i: below[m]): i)

  ccard(s: cfin) : nat = IF m = 0 THEN 0 ELSE ccardi(s, m-1) ENDIF

 END countable_cardinality
```

```
% finite_cardinality provides the fincard function, which gives the
% cardinality of a finite set whose base type is known to be finite.
% The f parameter gives an injective mapping from the naturals to the base
% type of the sets, and the m parameter gives an upper bound on the
% cardinality.  This form of cardinality is useful when the base type is
% finite, as it is then known that every set defined on that type is
% also finite.

finite_cardinality[T: TYPE, m: nat, f: (bijective?[below[m], T])] : THEORY
 BEGIN

  fincardi(s: set[T], i: below[m]) : RECURSIVE nat =
    (IF i = 0 THEN IF s(f(i)) THEN 1 ELSE 0 ENDIF
              ELSIF s(f(i)) THEN fincardi(s, i-1) + 1
              ELSE fincardi(s, i-1) ENDIF)
    MEASURE (LAMBDA (s: set[T]), (i: below[m]): i)

  fincard(s: set[T]) : nat = IF m = 0 THEN 0 ELSE fincardi(s, m-1) ENDIF

 END finite_cardinality
```

Number Properties

The properties of the numbers is expressed in the next two theories. The `numberaxioms` theory has the axioms, and the `numberprops` theory has useful properties that may be derived from the axioms. In general, it is not necessary to cite any of these formulas, as the decision procedures usually take care of them automatically. However, ocassionally a nonlinear arithmetic expression is beyond the capabilities of the prover, in which case these formulas may help. In addition, if desired, all proofs may be done from "first principles," which, although extremely tedious, may provide more insight or confidence in the proof. For large verification efforts, this is a ludicrous approach.

```
% number_axioms provides the usual commutativity, associativity, etc.
% axioms about the real numbers.  Note that many of these properties
% also hold for the rationals, integers, and natural numbers.  These
% properties are all known to the decision procedures of PVS, so should
% rarely need to be cited.

number_axioms : THEORY

 BEGIN

  % Field Axioms

  commutative_add: POSTULATE FORALL (x,y:real): x + y = y + x

  associative_add: POSTULATE FORALL (x,y,z:real):
        (x + y) + z = x + (y + z)

  identity_add: POSTULATE FORALL (x:real): x + 0 = x

  inverse_add: POSTULATE FORALL (x:real): x + -x = 0

  minus_add : POSTULATE  FORALL (x, y:real): x - y = x + -y

  commutative_mult: POSTULATE FORALL (x,y:real): x * y = y * x

  associative_mult: POSTULATE FORALL (x,y,z:real):
        (x * y) * z = x * (y * z)

  unique_identities: POSTULATE 0 /= 1

  identity_mult: POSTULATE FORALL (x:real): x * 1 = x

  inverse_mult: POSTULATE FORALL (x:nonzero_real): x * (1/x) = 1

  distributive: POSTULATE FORALL (x,y,z:real):
    x * (y + z) = (x * y) + (x * z)
```

```
  % Order Axioms

  posnum?((x : real)) : bool = x > 0

  posnum : TYPE = (posnum?)

  posnum_add_closed: AXIOM FORALL (x,y: posnum): x + y > 0

  posnum_mult_closed: AXIOM FORALL (x,y: posnum): x * y > 0

  posnum_neg: AXIOM FORALL (x: posnum): NOT posnum?(-x)

  trichotomy: AXIOM FORALL (x: real):
    x > 0 OR x = 0 OR 0 > x

  % Completeness Axiom

  upper_bound?((ub: real), (S: (nonempty?[real]))): bool =
    (FORALL (x: real): member(x,S) => x <= ub)

  least_upper_bound?((lub: real), (S: (nonempty?[real]))): bool =
    upper_bound?(lub,S) &
      FORALL (x: real): upper_bound?(x, S) => lub <= x

  real_complete: AXIOM FORALL (S: (nonempty?[real])):
    (EXISTS (y : real) : upper_bound?(y, S)) =>
    (EXISTS (x:real): least_upper_bound?(x,S))

 END number_axioms
```

```
% number_props contains useful properties about real numbers.  In
% general these are known to the decision procedures of PVS, but they
% may occasionally be necessary.

number_props : THEORY

 BEGIN

  strict_lt : LEMMA strict_total_order?(<)

  div_prop1: LEMMA FORALL (x,y:real),(z:nonzero_real):
        x * (y/z) = (x * y)/z

  div_cancel1: LEMMA FORALL (x:real),(y:nonzero_real):
        y * (x/y) = x

  div_cancel2: LEMMA FORALL (x:real),(y:nonzero_real):
        (x/y) * y = x
```

```
  cross_mult : LEMMA
     FORALL (x,y:real),(u, v:nonzero_real):
      ((x/u) = (y/v)) = (x * v = y * u)

  add_div : LEMMA
     FORALL (x,y:real),(u, v:nonzero_real):
      ((x/u) + (y/v)) = (x * v + y * u)/(u * v)

  minus_div : LEMMA
     FORALL (x,y:real),(u, v:nonzero_real):
      ((x/u) - (y/v)) = (x * v - y * u)/(u * v)

  both_sides : LEMMA
    FORALL (x,y:real),(z:nonzero_real):
      (z * x = z * y) => x = y

  cancel : LEMMA
    FORALL (x,y, z:real) :
       x = y => (z * x = z * y)

  div_distributes : LEMMA
      FORALL (x,y:real),(z:nonzero_real):
         (x + y)/z = (x/z) + (y/z)

  div_distributes_minus : LEMMA
      FORALL (x,y:real),(z:nonzero_real):
         (x - y)/z = (x/z) - (y/z)

  div_div : LEMMA
      FORALL (x: real),(y, z:nonzero_real):
        (x / (y / z)) = ((x * z) / y)

  div_div2 : LEMMA
      FORALL (x: real),(y, z:nonzero_real):
        ((x / y) / z) = (x / (y * z))

END number_props
```

Ordinals

The ordstruct datatype provides the constructible ordinals. These are the ordinal numbers below ε_0 ($=\omega^{\omega^{\omega^{\cdot^{\cdot^{\cdot}}}}}$). They are either zero or of the form $n\omega^\alpha+\beta$, where α and β are of type ordstruct, and n is a positive integer. As with ordinary polynomials, it is more convenient to work with ordstructs if they are given in canonical form, where terms of higher degree precede those of lower degree. This is the purpose of the ordinals theory, which defines the order < and defines the ordinal type to consist of those ordstructs that respect the order. See Section 3 for an example of the use of ordinals. More discussion may be found in [8, pp. 476–479].

```
% ordstruct provides the building blocks for defining the constructive
% ordinals.

ordstruct: DATATYPE
 BEGIN
  zero: zero?
  add(coef: posnat, exp: ordstruct, rest: ordstruct): nonzero?
 END ordstruct
```

```
% ordinals uses the ordstruct datatype to define the constructive
% ordinals, by providing an ordering on ordstruct and defining the
% ordinals to be those elements of type ordstruct that are in a
% canonical form with respect to the ordering.

ordinals: THEORY
 BEGIN
  i, j, k: VAR posnat
  m,n, o: VAR nat
  x, y, z: VAR ordstruct
  size: [ordstruct->nat] = ordstruct_rec[nat](0, (LAMBDA i, m, n: 1 + m+n))
  r, s, t, u, v, w: VAR ordstruct
  <: RECURSIVE [ordstruct, ordstruct -> bool] =
    (LAMBDA x, y:
      (CASES x OF
        zero: NOT zero?(y),
        add(i, u, v): CASES y OF
                     zero: FALSE,
                     add(j, z, w): (u<z) OR
                                   (u=z) AND (i<j) OR
                                   (u=z) AND (i=j) AND (v<w)
                  ENDCASES
     ENDCASES))
  MEASURE (LAMBDA x, y: size(x));

 >(u,v): bool = v < u;
 <=(u,v): bool = u < v OR u = v;
 >=(u,v): bool = v < u OR v = u;

 ordinal?: RECURSIVE pred[ordstruct] =
  (LAMBDA x:
    CASES x OF
     zero: TRUE,
     add(i, u, v): (ordinal?(u) AND ordinal?(v) AND
                     CASES v OF
                       zero: TRUE,
                       add(k,r,s): r < u
                     ENDCASES)
    ENDCASES)
  MEASURE size

 ordinal: TYPE = (ordinal?)
```

```
 irreflexive: LEMMA (FORALL (x:(ordinal?)): NOT x < x)

 antisym: LEMMA (FORALL (x,y:(ordinal?)): x < y IMPLIES NOT y < x)

 antisymmetric: LEMMA
    (FORALL (x, y: (ordinal?)): x <= y AND y <= x IMPLIES x = y)

 transitive: LEMMA
    (FORALL (x,y,z:(ordinal?)): x < y AND y < z IMPLIES x < z)

 ordinal_induction: AXIOM
    (FORALL (p: pred[ordinal]):
      (FORALL (x:ordinal): (FORALL (y:ordinal): y < x IMPLIES p(y))
                              IMPLIES p(x))
        IMPLIES (FORALL (x:ordinal): p(x)))

 well_founded_le : LEMMA
   well_founded?(LAMBDA (x, y: (ordinal?)) : x <= y)

END ordinals
```

Lists

The `list` datatype defines lists; see Section 8 for details. Theory `list_props` defines the `length` function, and `every`, which returns TRUE if the predicate is true for every element of the given list. The `lists` theory defines the `map` function, which maps a function over a list, producing a list.

```
% list provides the datatype definition for lists.

list[t:TYPE] : DATATYPE
 BEGIN
  null: null?
  cons (car: t, cdr :list) :cons?

 END list
```

```
% list_props provides the length function and the function every, which
% tests a predicate against every element of a list.

list_props[t:TYPE] : THEORY
 BEGIN
  length(l:list[t]): RECURSIVE nat =
    CASES l OF
      null: 0,
      cons(x,y): length(y) + 1
    ENDCASES
    MEASURE list_nat_rec(0, (LAMBDA (x: t), (n: nat): n + 1))

  every(p: pred[t], l: list[t]): RECURSIVE bool =
    CASES l OF
      null: true,
      cons(x, y): p(x) & every(p, y)
    ENDCASES
    MEASURE (LAMBDA (p: pred[t]), (l: list[t]): length(l))

 END list_props
```

```
% lists defines the map function, which maps a function over a list,
% returning a list.

lists[t1,t2:TYPE] : THEORY
 BEGIN
  map(f: [t1 -> t2], l: list[t1]): RECURSIVE list[t2] =
    CASES l OF
      null: null,
      cons(x,y): cons(f(x), map(f, cdr(l)))
    ENDCASES
    MEASURE (LAMBDA (f: [t1 -> t2]), (l: list[t1]): length(l))

 END lists
```

Maps and Filters

The `map_props` theory gives the commutativity properties of composition and `map`, for both sequences and lists.

```
% map_props gives the commutativity properties of composition and map,
% for both sequences and lists.

map_props[t1,t2,t3: TYPE] : THEORY
 BEGIN
  map_prop1: LEMMA
    (FORALL (f: [t1 -> t2]),(g: [t2 -> t3]),(s: sequence[t1]):
      map(g, map(f, s)) = map(g o f, s))

  map_prop2: LEMMA
    (FORALL (f: [t1 -> t2]),(g: [t2 -> t3]),(s: list[t1]):
      map(g, map(f, s)) = map(g o f, s))

 END map_props
```

Theory `filters` provides filtering functions for sets and lists. Filter functions for sequences cannot be defined in general, since sequences are infinite, and the predicate may only hold for a finite number of elements of the input sequence.

```
% filters defines filter functions for sets and lists, which take a set
% (list) and a predicate and return the set (list) of those elements
% that satisfy the predicate.

filters[t: TYPE] : THEORY
 BEGIN
  filter(s: set[t], p: pred[t]): set[t] =
    {x: t | s(x) & p(x)}

  filter(l: list[t], p: pred[t]): RECURSIVE list[t] =
    CASES l OF
      null: null,
      cons(x, y): IF p(x) THEN cons(x, filter(y, p)) ELSE filter(y, p) ENDIF
    ENDCASES
    MEASURE (LAMBDA (l: list[t]), (p: pred[t]): length(l))
 END filters
```

Cardinality Properties

Some useful properties regarding cardinalities.

```
% card_set provides a small set of standard lemmas
% regarding finite cardinality, filters, and sets.

card_set[T: TYPE, n:posnat, f:(bijective?[below[n],T])] : THEORY
 BEGIN

  IMPORTING finite_cardinality[T,n,f],
            filters[below[n]]

  fullset_fincard: LEMMA fincard(fullset[T]) = n

  fincard_non_empty: LEMMA
    (FORALL (m:setof[T]):
      (EXISTS (z:T): m(z)) IFF fincard(m) /= 0)

  fincard_filter: LEMMA
    (FORALL (m1,m2:setof[T]):
      fincard(filter(m1,m2)) <= fincard(m1))

  zero_fincard: LEMMA
    (FORALL (y:T),(m1,m2:setof[T]):
       m1(y) AND fincard(filter(m1,m2)) <= 0 IMPLIES NOT m2(y))

  remove_prop: LEMMA
    (FORALL (y,z:T),(m:setof[T]):
      m(z) AND (NOT y=z) IMPLIES remove(y, m)(z))

  fincard_remove: LEMMA
    (FORALL (z:T),(m:setof[T]):
      IF m(z)
      THEN fincard(remove(z, m)) = fincard(m) - 1
      ELSE fincard(remove(z, m)) = fincard(m)
      ENDIF)

  remove_comm: LEMMA
    (FORALL (z:T),(m1,m2:setof[T]):
      filter(remove(z, m1),m2) = remove(z, filter(m1,m2)))

 END card_set
```

References

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

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

[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.

[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.

[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.

[6] Joseph A. Goguen and Timothy Winkler. Introducing OBJ. Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1988.

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

[8] Stephen C. Kleene. *Introduction to Metamathematics.* North-Holland, 1952.

[9] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[10] 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.

[11] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

Index