SRI International

CSL Technical Report SRI-CSL-03-01 • March 7, 2003

The PVS Prelude Library

S. Owre and N. Shankar

Funded by NSF Grants CCR-0082560 and CCR-9712383, DARPA/AFRL Contract F33615-00-C-3043, and NASA Contract NAS1-20334.

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Abstract

The PVS Prelude Library is a collection of basic theories about logic, functions, predicates, sets, numbers, and other datatypes. The theories in the prelude library are visible in all PVS contexts, unlike those from other libraries that have to explicitly imported. These theories also illustrate various language features of PVS that are useful formalization aids.

Contents

Chapter 1

Introduction

The PVS prelude is a large body of theories that provides the infrastructure for the PVS typechecker and prover, as well as much of the mathematics needed to support specification and verification of systems. It is worthwhile looking at the prelude, as it also provides a rich source of examples, both for specification and proofs.

This report is intended to serve as a roadmap to the prelude and is meant to be read in conjunction with the prelude itself.[1] Broadly speaking, the prelude can be divided into the logic, functions, relations, induction, sets, numbers, sequences, sum types, quotient types, and mu-calculus. These are described below. Note that the prelude is built sequentially, and declarations must be given prior to their use. This means that the conceptual division is not strictly followed in the prelude.

[1]In PVS, `M-x view-prelude-file` shows the prelude, or it is available at `ftp://ftp.csl.sri.com/pub/pvs/libraries/prelude.pvs`. Contributors to the prelude libraries include Ricky Butler, Paul Miner, Bruno Dutertre, Damir Jamsek, Michael Holloway, Bart Jacobs, and Jerry James.

Chapter 2

Logic: Booleans, Equality, Quantifiers, and Conditionals

The first declarations are those of the the type `boolean` and the boolean operators. Theoretically, this could be done using datatypes, but the datatype mechanism itself relies on the boolean type, so they are defined as uninterpreted types and functions. Equality, disequality, and `IF` operators are declared polymorphically, using types as theory parameters.

2.1 `booleans`

The theory `booleans` introduces the nonempty type `boolean` (or, `bool`, for short) with elements `TRUE` and `FALSE`, and the propositional operators for conjunction, `AND` or &, disjunction, `OR`, implication, `IMPLIES` or `=>`, converse implication, `WHEN`, and equivalence, `IFF` or `<=>`.

Some axioms about these operators are given as `postulates` in the theory `boolean_props` below to indicate that these properties are actually built-in as primitive inference rules of the PVS proof checker. The declarations given here provide the context for typechecking PVS formulas.

2.2 `equalities`

A *function* is a symbol of type `[D -> R]`, here `D` is the domain type and `R` is the range type. A *predicate* is a function whose range type is `boolean`. The theory equalities takes a single type parameter `T` and declares the equality symbol `=` as a binary *predicate* over this type. Some properties of equality are given as postulates in the theory `eq_props` below.

2.3 notequal

The theory notequal also takes a type parameter T and introduces the binary disequality predicate /=.

2.4 if_def

The theory if_def takes a type parameter T and declares a conditional operator IF which takes a three arguments: a boolean *test* argument, and a *then* and *else* part that are both of type T. The conditional operator has a mix-fix syntax as IF test THEN expr1 ELSE expr2 ENDIF. The axioms for the IF operator are unspecified and built-in as primitive inference rules in the PVS proofchecker.

2.5 boolean_props

The operators introduced in the theory boolean are defined here in terms of equality and IF. The properties have already been incorporated into the primitive inference rules of PVS proofchecker like FLATTEN and SPLIT.

2.6 xor_def

The exclusive-or connective or boolean disequality, XOR, is defined in the theory xor_def.

2.7 quantifier_props

The theory quantifier_props introduces the existential and universal quantifiers with respect to a type parameter t. These are binding operations so that the occurrences of x and y in x /= y are bound in EXISTS x, y: x /= y.

2.8 defined_types

The theory defined_types introduces pred[t] and setof[t] as abbreviations for the predicate type [t -> bool].

2.9 exists1

The unique existence quantifier is defined in the theory in the theory exists1. It illustrates how a second-order operator like exists1 can be turned into a binding operator as exists1!.

2.10 equality_props

Postulates about equality and `IF` are given in `equality_props`. The congruence postulate is given in the theory `functions` below.

2.11 if_props

A couple of simple properties of conditionals are given in `if_props`. These are given in a separate theory and not in `if_def` since two type parameters are required.

Chapter 3

Functions

As PVS is based on higher-order logic, functions are the basis for much of the expressive power of the language. There are several theories that develop definitions and lemmas, including injective, surjective, and bijective functions, the restriction and extension conversions, and the K_conversion, which is important for triggering lambda conversions, described in the PVS language manual [4]. The function inverse is defined in two different ways. Theory `function_inverse_def` defines relations that hold between a function and its inverse, without actually defining an inverse. Theory `function_inverse` defines the inverse in terms of the `epsilon` function, which requires the domain type to be nonempty. Theory `function_inverse_alt` weakens this restriction by including an assumption that either the domain is nonempty or the range is empty. Function image is defined, and a number of theories developing properties of functions are provided.

3.1 functions

The `functions` theory 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. However, it doesn't handle dependent function types, since the domain and range cannot be given as independent parameters.

Extensionality and congruence postulates are given, as well as the related `eta` lemma. The `injective?`, `surjective?`, and `bijective?` predicates are defined, as well as judgements relating them. The `domain` and `range` types are defined. The `graph` function converts a function to a relation. The `preserves` (`inverts`) predicate holds for a given function `f` and relations $R_{\mathtt{D}}$, $R_{\mathtt{R}}$ over the domain and range, respectively, if $xR_{\mathtt{D}}y$ then $f(x)R_{\mathtt{R}}f(y)$ $(f(y)R_{\mathtt{R}}f(x))$.

3.2 functions_alt

This simply redefines the `preserves` and `inverts` functions in a theory where the $R_{\mathtt{D}}$ and $R_{\mathtt{R}}$ relations are theory parameters. This is useful for working with a fixed pair of relations; the instance may be specified in an importing, rather than in each use of these functions.

3.3 restrict

`restrict` is the restriction operator on functions, allowing a function defined on a supertype `T` to be turned into a function over a subtype `S`. It is made a *conversion* that is automatically inserted by the typechecker to correct type mismatches. The fact that the restriction of an injective function is injective is noted as a lemma and a *judgement*. The typecker annotates a term by the more refined type information provided by such judgements. Thus, whenever the `restrict` operation is applied to a function that is known to be injective, the resulting function is also known to be injective.

3.4 restrict_props

This simple theory just notes that `restrict` is the identity when the subtype is not a proper subtype.

3.5 extend

The function `extend` is the inverse of `restrict`. The difference is that there is only one possible restriction, whereas in general there are a large number of possible extensions to a given function. The form of extension provided by this theory just uses a provided default element of the range type that all elements of the domain extension are mapped to. The `restrict_extend` lemma says that the restriction of an extension is the identity.

3.6 extend_bool

Though `extend` is generally not useful as a conversion, when the range type is boolean it makes sense to make the default value `false`. This allows, for example, predicates on natural numbers to be treated as predicates on integers (equivalently, sets of natural numbers as sets of integers). This theory simply introduces this conversion.

3.7 extend_props

extend_props provides the lemma that extending a function from a given domain type to the same type is the identity. This usually comes about because of theory instantiation, and the typechecker has this rule built in, so that it is not needed in general.

3.8 extend_func_props

This theory simply provides the judgement that the extension of a surjective function is surjective.

3.9 K_conversion

The K combinator, called K_conversion, is defined here as $\lambda x.\lambda y.x$. When enabled as a conversion, it triggers *lambda-conversions*, as described in the PVS language reference [4]. This is useful as a way of formalizing states and computations over states within higher-order logic. The conversion is not enabled by default, because the typechecker frequently finds ways to mask type errors by applying this, leading to unintended specifications that are not noticed until proofs are attempted.

3.10 K_props

This theory provides judgements that K_conversion preserves subtypes.

3.11 identity

This defines the parametric identity function as I, id or identity. Any of the identifiers I, id, or identity may be used. All three are declared to be bijective.

3.12 identity_props

This theory provides judgements that the identity function preserves subtypes.

3.13 function_inverse_def

This provides the function inverse relations, but does not actually define the inverse function, see function_inverse and function_inverse_alt for two possible definitions. This theory defines the left-, right-, and two way inverse relations, provides a number of lemmas relating these to other functional relations, for example, surjectivity and injectivity, and provides existence lemmas that are enough to discharge the assumptions that need to be discharged if the `function_inverse_alt` theory is used.

3.14 function_inverse

This theory defines the `inverse` function in terms of the `epsilon` function (see `epsilons` below), and hence requires that the domain type parameter be nonempty. The rest of the theory relates inverse to other functional properties such as injectivity and surjectivity, and provides corresponding judgements.

3.15 function_inverse_alt

This theory provides an alternative definition for inverse, called `inverse_alt`, but with fewer restrictions: the domain is nonempty or the range is empty. When the domain is known to be nonempty, `function_inverse` is generally easier to work with. Judgements are provided for `inverse_alt`. The existence lemmas of `function_inverse_def` may be usedful in discharging the assumptions that result from using this theory.

3.16 function_image

This theory provides the `image` and `inverse_image` functions, in both curried and uncurried forms. `inverse_image` is not the same as `inverse`; it is defined for all functions, not just injections, and returns a set. Several lemmas are provided relating these to various set operations.

3.17 function_props

This theory defines the functional composition operator `o`, provides the judgements that composition of pairs of injective, surjective, and bijective functions are respectively injective, surjective, and bijective, and states lemmas that relate composition to the image, preserves, and inverts functions.

3.18 function_props_alt

`function_props_alt` gives judgements relating composition to the `preserves` and `inverts` operators. The difference is that this theory has the relations as parameters rather than as variables. Thus this theory is easier to use if the relations are fixed.

3.19 function_props2

This theory simply states that composition is associative. It needs a separate theory in order to provide enough type parameters.

3.20 operator_defs

The `operator_defs` theory provides the predicates associated with operators, e.g., the plus and times operators associated with a ring. It provides the predicates `commutative?`, `associative?`, `left_identity?`, `right_identity?`, `identity?`, `has_identity?`, `zero?`, `has_zero?`, `inverses?`, `has_inverses?`, and `distributive?`.

3.21 function_image_aux

This theory defines judgements and lemmas that show that the image of a function on a finite set is finite, that the cardinality of the image of a set is less than or equal to that of the set, and equal when the function is injective, and that the image of an injective function is equipotent (i.e., there is a bijection) to the domain.

3.22 function_iterate

Provides a way to iterate a function application n times, i.e.,

$$f^n(x) = \overbrace{f(\cdots(f}^{n}(x))).$$

Lemmas such as $f^m \circ f^n = f^{m+n}$ are also provided.

3.23 PartialFunctionDefinitions

Two representations of partial functions are described, and shown to be isomorphic. `SubsetPartialFunction` is defined as a dependent record type, and `LiftPartialFunction` is defined on the lifted range type. In practice, the formulation based on `lift` is more convenient, because definitions are easier and fewer TCCs are generated.

3.24 PartialFunctionComposition

This theory defines composition operators for the partial functions defined above.

Chapter 4

Relations

Relations play an important role in specifications of systems, and the prelude provides many useful definitions and properties, including reflexivity, equivalence, preorders, partial orders, well orderings, and least upper bounds and greatest lower bounds.

4.1 `relations`

The `relations` theory defines relational predicates, including `reflexive?`, `irreflexive?`, `symmetric?`, `antisymmetric?`, `connected?`, `transitive?`, and `equivalence?`.

4.2 `orders`

The `orders` theory defines the usual ordering predicates: `preorder?`, `partial_order?`, `strict_order?`, `dichotomous?`, `total_order?`, `linear_order?`, `trichotomous?`, `strict_total_order?`, `well_founded?`, `well_ordered?`, `upper_bound?`, `lower_bound?`, `least_upper_bound?`, and `greatest_lower_bound?`. Numerous judgements relating these predicates are also provided.

4.3 `orders_alt`

The `orders_alt` theory defines `upper_bound?`, `least_upper_bound?`, `lower_bound?`, and `greatest_lower_bound?`, but with theory parameters providing the order and subset. This is useful when the order and subset are fixed.

4.4 `restrict_order_props`

This theory provides a set of judgements that the restriction of certain relations to a subtype still satisfy the relations. For example, the restriction of a reflexive relation is reflexive.

4.5 extend_order_props

This theory provides a set of judgements stating that the extension of certain relations to a supertype still satisfy the relations. For example, the extension of an irreflexive relation is irreflexive. The extension in this case is such that the relation is false on elements of the extension.

4.6 relation_defs

This theory defines more general relations between two possibly distinct types. It defines the operators `domain`, `range`, `image`, `preimage`, `postcondition`, `precondition`, `converse`, `isomorphism?`, `total?`, and `onto?`.

4.7 relation_props

This defines the relational composition operator `o` and judgements and lemmas relating to it.

4.8 relation_props2

Proves associativity of the relational composition operator. This is needed in a separate theory in order to provide the right number of types.

4.9 relation_converse_props

This theory provides a set of judgements that state that the converse of certain relations satisfies the relation. For example, the converse of a reflexive relation is reflexive.

Chapter 5

Induction

Induction is important in proving properties of systems. The prover `induct` rule can make use of these induction lemmas. There are other induction lemmas in the prelude, that are all variants of natural number induction; see Section 7.

5.1 `wf_induction`

This defines the well-founded induction schema `wf_induction`.

5.2 `measure_induction`

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

Chapter 6

Sets

Sets in PVS (as in most higher-order logics) are represented as predicates, i.e., functions from a given type to `boolean`. Membership is thus simply application of a set to an element; the element belongs to the set if the application returns `true`. This means that a set may be given in either of the equivalent forms `{x: T | p(x)}` or `LAMBDA (x: T): p(x)`.

All the usual set theoretic operators are available, e.g., `union`, `intersection`, `difference`, and `powerset`. There are also `Union` and `Intersection`, for use on sets of sets, and `IUnion` and `IIntersection` for indexed sets.

A notion of ordinal is defined, based on Cantor normal form. This only produces ordinals up to ϵ_0, transfinite induction is not possible with this. To do transfinite induction, set theory should be developed axiomatically within a single (non-parametric) theory. Since the primary use of PVS is for specification of systems, this has not been done. However, finite sets are very important, and the basic definitions and lemmas are in the prelude. There is also a `finite_sets` library that builds on this. Infinite sets are also defined, though there is no cardinality function for these.

6.1 `epsilons`

`epsilons` provides the Hilbert epsilon function. This acts as a "choice" function. The domain type must be nonempty, but the predicate need not be. Given a predicate over the parameter type, `epsilon` produces an element satisfying that predicate if one exists, and otherwise produces an arbitrary element of that type. Note that because the type parameter is given as nonempty, a nonempty TCC may be generated when this is used.

6.2 `sets`

Sets are modeled as predicates. The `sets` theory defines the usual set operators `member`, `empty?`, `emptyset`, `nonempty?`, `full?`, `fullset`, `subset?`, `strict_sub-`

set?, union, intersection, disjoint?, complement, difference, symmetric_difference, every, some, singleton?, singleton, add, remove, choose, the, rest, powerset, Union, and Intersection.

6.3 sets_lemmas

Several lemmas are provided about the operators defined in sets. These generally follow the lemmas and exercises provided in any introductory text on Set Theory (for example, Halmos [1]).

6.4 indexed_sets

This defines the IUnion ($\bigcup_{i \in I} A_i$) and IIntersection ($\bigcap_{i \in I} A_i$) operations, and lemmas about them.

6.5 finite_sets

The finite_sets theory develops finite sets as a subtype of sets for which there is an injection to a prefix of the natural numbers. Cardinality is defined, and several lemmas and judgements are provided (for example, the union of finite sets is finite).

6.6 restrict_set_props

This theory provides a set of judgements and lemmas that the restriction of a finite set to a subtype is still finite, and its cardinality is smaller.

6.7 extend_set_props

This theory provides a set of judgements and lemmas relating to the extension of a set. For example, that the extension of a finite set is finite, and the cardinality is the same.

6.8 ordstruct and 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. This essentially works with canonical representatives of an equivalence class. Note that although this is a large ordinal number, it is still countable (has cardinality $\aleph_0$). More discussion about ε_0 may be found in [2, pp. 476–479].

6.9 `infinite_sets_def`

This defines the notion of an infinite set and provides theorems and judgements similar to those for finite sets. No notion of cardinality is given, however.

6.10 `finite_sets_of_sets`

This theory gives several judgements such as that the powerset of a finite set is finite, and the union of a finite number of finite sets is finite.

Chapter 7

Numbers

The usual practice in mathematics is to start from the natural numbers (e.g., the Peano axioms), and build integers as equivalence classes of pairs of nats, rationals as equivalence classes of pairs of integers, and reals as equivalence classes of Cauchy sequences or Dedekind cuts. This is very nice for foundations, but cumbersome to use in practice. In PVS we take the axiomatic approach and reverse this; the universal `number` type is given, and the rest are subtypes of it. The `number` type is completely uninterpreted, though it contains all the numerals. The `number_field` type introduces the field operators and axioms. The reason for introducing it is that, for example, the complex numbers may be introduced as a subtype of `number_field` with an axiom that it contains all reals plus new constant i, and the operators may simply be used, without having to extend them. This would not work with the reals, as they include an ordering that is incompatible with the complex numbers. Of course, other number systems could be inserted as well, for example the nonstandard reals.

The axioms used for the number_fields and reals were taken from Royden [6]. Note that many of the real axioms and lemmas are already "known" to the decision procedures, but nonlinear properties frequently require the use of the axioms or lemmas. The `real_props` theory is useful in this regard, and using it as an `auto-rewrite-theory` can make proofs a lot simpler.

7.1 `numbers`

This provides the top `number` type, of which all other number types are subtypes. All of the numerals are implicitly of this type.

7.2 `number_fields`

`number_fields` defines the type `number_field`, the field operations `+`, `-` (unary and binary), `*`, and `/`, and the field axioms. Note that any field containing the reals (e.g., nonstandard reals, complex numbers) could be made a subtype of this. In the following example, there is no need to create new declarations for the field operators, and no representations are needed; the real numbers are already complex numbers. Note that the decision procedures are still sound, because they only interpret these operators when the operands are known to be real.

```
complex: THEORY
 BEGIN
  complex: NONEMPTY_TYPE FROM number_field
  real_are_complex: AXIOM FORALL (x: real): complex_pred(x)
  JUDGEMENT real SUBTYPE_OF complex
  nonzero_complex: NONEMPTY_TYPE
    = {c: complex | c /= 0} CONTAINING 1
  nzcomplex: NONEMPTY_TYPE = nonzero_complex

  i: complex
  i_ax: AXIOM i * i = -1

  rep_exists: AXIOM
    FORALL (c: complex): EXISTS (x, y: real): c = x + y*i
  rep_unique: AXIOM
    FORALL (x1, x2, y1, y2: real):
      x1 + y1*i = x2 + y2*i <=> (x1 = x2 & y1 = y2)
 END complex
```

7.3 reals

The `reals` theory defines the `real` type as a subtype of `number_field`, defines closure judgements for the field operators, and adds the order operators `<`, `<=`, `>`, and `>=`. The numerals implicitly belong to this type.

7.4 real_axioms

This theory simply gives the order axioms for the reals: the sum and product of positive reals is positive, the negation of a positive real is not positive, and every real is greater than, equal to, or less than 0.

7.5 bounded_real_defs

The `bounded_real_defs` theory provides definitions for `upper_bound?`, `lower_bound?`, `least_upper_bound?`, and `greatest_lower_bound?`, then gives the completeness axiom `real_complete` for the reals: every nonempty set with an upper bound has a least upper bound. The corresponding lemma `real_lower_complete` for lower bounds is also provided.

`bounded_above?`, `bounded_below?`, `bounded?`, `lub`, and `glb` are also defined along with some related lemmas.

7.6 `bounded_real_defs_alt`

This theory provides alternative definitions for `upper_bound?`, `lower_bound?`, `least_upper_bound?`, and `greatest_lower_bound?` where the nonempty set is provided as a theory parameter. This is useful when the set is fixed.

7.7 `real_types`

The `real_types` theory defines useful subtypes of reals: `nonneg_real`, `nonpos-_real`, `posreal`, and `negreal`, and provides several judgements relating these and the field operators.

7.8 `rationals`

This theory defines the rationals as an uninterpreted subtype of real, and provides closure judgements for the field operators. The numerals implicitly belong to the rationals.

7.9 `integers`

`integers` defines the integer type, along with the `upfrom`, `above`, `nonneg_int`, `nonpos_int`, `posint`, and `negint`, `subrange`, `even_int`, and `odd_int` types. It provides lots of judgements. The numerals implicitly belong to the integers.

7.10 `naturalnumbers`

`naturalnumbers` defines the natural number type (also known as the whole numbers) and the `upto` and `below` types. `succ`, `pred`, and natural number minus (, sometimes called *monus*) are defined, and finally weak and strong natural number induction lemmas are given. The numerals implicitly belong to the natural numbers.

7.11 `min_nat`

This theory defines the minimum `min` of a set of natural numbers.

7.12 real_defs

real_defs defines the sign function sgn, absolute value abs, maximum and minimum functions max and min, and several judgements involving these.

7.13 real_props

This theory provides dozens of lemmas about real numbers. Many of them are especially useful in dealing with nonlinear arithmetic, which are (necessarily) incomplete in the decision procedures. This theory can be used in the auto-rewrite-theory prover command, which often makes proofs involving real number arithmetic simpler.

7.14 rational_props

rational_props gives the axiom that any rational number is the quotient of two integers, and lemmas stating the density of the rationals.

7.15 integer_props

This provides several lemmas about integers and natural numbers, including specialized least upper bound (lub) and greatest lower bound (glb) properties.

7.16 floor_ceil

floor_ceil defines the floor and ceiling functions, and gives several lemmas and judgements pertaining to them.

7.17 exponentiation

exponentiation provides the definitions expt and ^. expt multiplies a real by itself the number of times specified, where 0 times returns a 1 (thus expt(0,0) = 1). ^ is defined in terms of expt to work for integers, but in this case if the integer is negative then the real argument must be nonzero; this leads to a dependent type. Several properties and judgements are also provided.

7.18 euclidean_division

This defines the mod function, and the Euclidean algorithm properties are given declaratively.

7.19 divides

This defines the `divides` relation between integers and provides lemmas and judgements accordingly.

7.20 modulo_arithmetic

This defines the `rem` and `ndiv` functions, and proves several lemmas about these operations.

7.21 subrange_inductions

This provides induction lemmas for the `subrange` type, suitable for use in the prover induction commands.

7.22 bounded_int_inductions

This theory provides induction lemmas for the `upfrom` and `above` types, suitable for use in the prover induction commands.

7.23 bounded_nat_inductions

This theory provides induction lemmas for the `upto` and `below` types, suitable for use in the prover induction commands.

7.24 subrange_type

This theory defines the `subrange` type in a parameterized theory, mostly for backward compatibility.

7.25 int_types

This just defines the `upfrom` and `above` types in a parameterized theory, mostly for backward compatibility.

7.26 nat_types

This theory defines the `upto` and `below` types in a parameterized theory, mostly for backward compatibility.

7.27 `nat_fun_props`

Special properties of injective, surjective, and bijective functions over the natural numbers.

7.28 `lex2`

`lex2` provides a lexical ordering for pairs of natural numbers. This illustrates the use of ordinals.

7.29 `exp2`

This theory defines the `exp2` function, which is simpler to use than `expt` for defining the bitvector theories.

Chapter 8

Sequences, lists, strings, and bitvectors

Sequences, finite sequences, lists, strings and bitvectors are treated in this chapter. Strings are built from characters, which are in this chapter.

Bit vectors are defined parameterized by the word size. The basic operations and their properties are given in the prelude. More extensive development is provided with the `bitvector` library.

8.1 `sequences`

`sequences` provides the polymorphic sequence type `sequence`, as a function from natural numbers to the base type. The usual sequence functions `nth`, `suffix`, `first`, `rest`, `delete`, `insert`, and `add` are also provided. Note that these are infinite sequences, and do not contain finite sequences as a subtype.

8.2 `seq_functions`

`seq_functions` defines the `map` function that generates a new sequence by applying a given function pointwise over the input sequence.

8.3 `finite_sequences`

Finite sequences are defined as a dependent record type `finite_sequence`, with the `length` as the first field and a `seq` as a function from the natural numbers below `length` to the base type. The `emptyseq` is defined, and a conversion is provided that allows a finite sequence to be applied to an index directly, without having to extract the `seq`. Composition `o` and concatenation `^` are defined. The `extract1` conversion is provided, that lets a sequence of length 1 to be treated as the single element. Finally the associativity of composition lemma is provided.

8.4 list

This defines the `list` datatype, with constructors `null` and `cons`, recognizers `null?` and `cons?`, and `cons` accessors `car` and `cdr`. See the PVS langauge manual [4] or the PVS datatype report [5] for details.

8.5 list_props

`list_props` provides the `length`, `member`, `nth`, `append`, and `reverse` functions. Several related lemmas are given.

8.6 map_props

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

8.7 filters

`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. Both the curried and uncurried forms are given.

8.8 list2finseq

This theory defines conversion function `lsit2finseq` from lists to finite sequences, and the inverse conversion, `finseq2list`.

8.9 list2set

This theory provides a conversion function `list2set` from lists to sets. Note that the other direction is not defined, though it could be, through the use of a choice function.

8.10 disjointness

The `disjointness` theory defines the `pairwise_disjoint?` function. This allows pairwise disjointness to be stated more succinctly.

8.11 character

The `character` datatype follows the ASCII control codes, of which only the first 128 are defined. This is used as the base type for strings. Note that because of the extend1 conversion, there is no need for special syntax for characters, for example, "f" = char(102) is type correct, and easily proved.

8.12 strings

The strings theory introduces the char type and defines the type string as a finite sequence of chars. The string_rep lemma shows how strings are represented internally. The other lemmas are useful for rewriting. This theory is useful to auto-rewrite with, but make sure that list2finseq is not also an auto-rewrite rule.

8.13 bit

A bit is a boolean, a nbit is either 0 or 1. The b2n conversion allows boolean values to be treated as nbits.

8.14 bv

The bv theory defines the bitvector type bvec, the useful bitvector constants bvec0 and bvec1, and the fill function and bit extraction operator ^.

8.15 bv_cnv

This simply defines the fill[1] function to be a conversion.

8.16 bv_concat_def

This theory defines the concatenation operator o for bitvectors.

8.17 bv_bitwise

Defines bit-wise logical operations OR, AND, IFF, NOT, and XOR on bit vectors, and provides some lemmas relating them to bit extraction.

8.18 bv_nat

Provides functions bv2nat and nat2bv that map bitvectors to natural numbers and vice versa. Several related lemmas are provided.

8.19 `empty_bv`

Defines the empty bitvector `empty_bv`.

8.20 `bv_caret`

The extractor operation ^ decomposes a bvec into smaller bit vectors. A few lemmas are also provided.

Chapter 9

Sum types

9.1 `lift`

The `lift` datatype adds a bottom element to a given base type. This is useful for defining partial functions as seen in the `PartialFunctionDefinitions` theory.

9.2 `union`

The `union` datatype provides a way of doing binary coproducts (also known as sums or cotuples). This is here mostly for backward compatibility, as the cotuple type constructor, introduced in PVS 3.0, removes the need for this type.

Chapter 10

Quotient types

Quotient types are important in mathematics, where it is common to introduce an equivalence relation and define a new structure as a set of equivalence classes, with operations "lifted" to the new structure. This is provided in the following theories. Note that one imoprtant use of this is in theory interpretations, where one often wants to interpret a type as an equivalence class. See the PVS Language Manual [4] or the PVS Theory Interpretations report [3].

10.1 `EquivalenceClosure`

This theory provides the higher order definition of equivalence relation closure `EquivalenceClosure` and several lemmas.

10.2 `QuotientDefinition`

`QuotientDefinition` defines the equivalence class function `EquivClass`, the equivalence class representative function `repEC`, the `Quotient` type, and the quotient type representative function `rep`. `quotient_map` is similar to `EquivClass`, but with a different range type, making it surjective. The `ECQuotient` type and `ECquotient_map` function are defined over arbitrary relations using `EquivalenceClosure`.

10.3 `KernelDefinition`

The `EquivalenceKernel` relation is defined on a function, and preserves lemmas are provided.

10.4 `QuotientKernelProperties`

This theory provides lemmas and judgements relating `EquivalenceKernel`, `Quotient`, and `quotient_map`.

10.5 `QuotientSubDefinition`

This provides `QuotientSub` and `quotient_sub_map`. These are restrictions of `Quotient` and `quotient_map` to a subtype.

10.6 `QuotientExtensionProperties`

This defines the `lift` function that lifts a function to a quotient. It does it using `QuotientSub`, in order to handle restricted functions properly.

10.7 `QuotientDistributive`

This theory makes clear that quotients commute with products: there is an isomorphism

$$[X/S, Y] \simeq [X, Y]/EqualityExtension(S)$$

given by the canonical map (from right to left). Such distributivity results can be used to define functions with several parameters on a quotient. In the presence of function types, this can also be done via Currying. The result is included here mainly as a test for the formalisation of quotients.

10.8 `QuotientIteration`

In this theory it will be shown how successive quotients can be reduced to a single quotient:

$$(X/S)/R \simeq X/action(S)(R)$$

again via the canonical map.

Chapter 11

Mu-calculus and CTL

These theories define predicate transformers, monotonicity, and the `mu` and `nu` operators as the least and greatest fixed points. The Computation Tree Logic (CTL) is then defined in terms of the mu-calculus. Note that the model checker built into PVS is based on the mu-calculus. Various forms of fairness are also provided.

11.1 mucalculus

This defines the `predicate_transformer` type, `monotonic?`, `fixpoint?`, `lfp?`, and `gfp?` predicates, and the `glb`, `lub`, `lfp`, `gfp`, `mu`, and `nu` functions. The induction lemmas `lfp_induction` and `gfp_induction` are also provided.

11.2 ctlops

This defines the basic CTL temporal operators `EX`, `EG`, `EU`, `EF`, `AX`, `AF`, `AG`, and `AU` in terms of the mu-calculus. No fairness is built in.

11.3 fairctlops

Fair versions of CTL operators where `fairAG(N, f)(Ff)(s)` means `f` always holds along every N-path from `s` along which the fairness predicate `Ff` holds infinitely often. This is different from the usual linear-time notion of fairness where the strong form asserts that if an action A is enabled infinitely often, it is taken infinitely often, and the weak form asserts that if any action that is continuously enabled is taken infinitely often.

11.4 `Fairctlops`

Fair versions of CTL operators with lists of fairness conditions. The expression `FairAG(N,f)(Fflist)(s)` means `f` always holds on every N-path from `s` along which each predicate in `Fflist` holds infinitely often.

Bibliography

[1] Paul R. Halmos. *Naive Set Theory.* The University series in undergraduate mathematics. Van Nostrand, 1960. Republished by Springer-Verlag in 1974 in the Undergraduate texts in mathematics series.

[2] S. C. Kleene. *Introduction to Metamathematics.* North-Holland, Amsterdam, 1952.

[3] S. Owre and N. Shankar. Theory interpretations in pvs. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001. Available at `http://pvs.csl.sri.com/doc/interpretations.html`.

[4] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, December 2001. Available at `http://pvs.csl.sri.com/doc/manuals.html`.

[5] Sam Owre and Natarajan Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Extensively revised June 1997. Available at `http://pvs.csl.sri.com/doc/manuals.html`. Also available as NASA Contractor Report CR-97-206264.

[6] H. L. Royden. *Real Analysis.* The Macmillan Company, New York, second edition, 1968.