Abstract Datatypes in PVS[1]

S. Owre N. Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025
Phone: (415)859-5272
{owre, shankar}@csl.sri.com
URL: `http://www.csl.sri.com/sri-csl-fm.html`

Technical Report CSL-93-9R
December 1993, Substantially Revised June 1997

[1]The development of the initial version of PVS was funded by internal research funding from SRI International. More recent versions of PVS have been developed with funding from ARPA, AFOSR, NASA, NSF, and NRL. Support for the preparation of this document came from the National Aeronautics and Space Administration Langley Research Center under Contract NAS1-18969. This report is a revised and updated version of Technical Report SRI-CSL-93-09.

Abstract

PVS (Prototype Verification System) is a general-purpose environment for developing specifications and proofs. This document deals primarily with the abstract datatype mechanism in PVS which generates theories containing axioms and definitions for a class of recursive datatypes. The concepts underlying the abstract datatype mechanism are illustrated using ordered binary trees as an example. Binary trees are described by a PVS abstract datatype that is parametric in its value type. The type of ordered binary trees is then presented as a subtype of binary trees where the ordering relation is also taken as a parameter. We define the operations of inserting an element into, and searching for an element in an ordered binary tree; the bulk of the report is devoted to PVS proofs of some useful properties of these operations. These proofs illustrate various approaches to proving properties of abstract datatype operations. They also describe the built-in capabilities of the PVS proof checker for simplifying abstract datatype expressions.

Contents

1 Introduction

PVS is a specification and verification environment developed at SRI International.[1] Several documents describe the use of PVS [OSR93]; this document explains the PVS mechanisms for defining and using abstract datatypes.[2] It describes a PVS specification for the data structure of ordered binary trees, defines various operations on this structure, and contains PVS proofs of some useful properties of these operations. It also describes various other data structures that can be captured by the PVS abstract datatype mechanism, and documents the built-in capabilities of the PVS proof checker for simplifying abstract datatype expressions. The exposition does assume some general familiarity with formal methods but does not require any specific knowledge of PVS.

PVS provides a mechanism for defining abstract datatypes of a certain class. This class includes all of the "tree-like" recursive data structures that are *freely generated* by a number of *constructor* operations.[3] For example, the abstract datatype of lists is generated by the constructors `null` and `cons`. The abstract datatype of stacks is generated by the constructors `empty` and `push`. An unordered list or a *bag* is an example of a data structure that is not freely generated since two different sequences of insertions of elements into a bag can yield equivalent bags. The queue datatype is freely generated but is not considered recursive in PVS since the accessor `head` returning the first element of the queue is not an inverse of the `enqueue` constructor. This means that the queue datatype must either be explicitly axiomatized or implemented using some other datatype such as the list or stack datatype.

At the semantic level, a recursive datatype introduces a new type constructor that is a solution to a recursive type equation of the form $T = \tau[T]$. Typically, the recursive occurrences of the type name T on the right-hand side must occur only *positively* (as defined in Section 2.1) in the type expression $\tau[T]$ and the datatype is the least solution to the recursion equation. For example, the datatype of lists of element type A is the least solution to the type equation $T = \{\texttt{null}\} + A \times T$, where $+$ is the disjoint union operation and the $\times$ operation returns the Cartesian product. The minimality of lists datatype yields

[1]PVS is freely available and can be obtained via FTP from `/pub/pvs/` through the Internet host `ftp.csl.sri.com`. The URL `http://www.csl.sri.com/pvs.html` provides access to PVS-related information and documents.

[2]The PVS abstract datatype mechanism is still evolving. Some of the contemplated changes could invalidate parts of the description in this report. This report itself updates SRI CSL Technical Report CSL-93-9 so that it is accurate with respect to the alpha version of PVS 2.1. Future versions of the report will be similarly revised to maintain accuracy.

[3]The abstract datatype mechanism of PVS is partly inspired by the *shell* principle used in the Boyer-Moore theorem prover [BM79]. Similar mechanisms exist in a number of other specification and programming languages.

a structural induction principle asserting that any list predicate P, if P is closed under the list datatype operations, i.e., where $P(\texttt{null})$ and $\forall x, l\colon P(l) \supset P(\texttt{cons}(x, l))$, then P holds of all lists. The induction principle also yields a structural recursion theorem asserting that a function that is defined by induction on the structure is total and uniquely defined. By the semantic definition of lists, the equality relation on the lists datatype is also the least equality where the constructor `cons` can be regarded as a congruence. The minimality of the equality relation asserts that the constructor `cons` is an injective operation from $A \times$ `list` to `list`. As a consequence of the minimality of equality on the datatype, one can define accessor functions such as `car` and `cdr` on lists constructed using `cons`, derive extensionality principles, and define functions by case analysis on the constructor. The PVS datatype mechanism is used to generate theories introducing the datatype operations for constructing, recognizing, and accessing datatype expressions, defining structural recursion schemes over datatype expressions, and asserting axioms such as those for extensionality and induction.

The datatype of lazy lists or streams is also generated by the same recursion scheme using the constructors `null` and `cons` but it is a co-recursive datatype (or a co-datatype) rather than a recursive datatype in that it is the greatest solution to the same recursion equation corresponding to lists. PVS does not yet have a similar mechanism for introducing co-datatypes, and this would be a useful extension to the language. Such a theory of sequences has been formalized in PVS by Hensel and Jacobs [HJ97] (see also the URL: `http://www.cs.kun.nl/~bart/sequences.html`).

PVS is a specification language with a set-theoretic semantics. Types are therefore interpreted as sets of elements and a function type `[A -> B]` is interpreted as the set of all total maps from the set corresponding to `A` to that for `B`. The use of set-theoretic semantics leads to some important constraints on the form of recursive definitions that can be used in PVS datatype declarations.

In Section 2, we first present the declaration for the `list` datatype to convey the syntactic restrictions on such datatype declarations. The outcome of such datatype declarations in terms of generated theories is explained in detail for the datatype of binary trees in Section 3. In Section 4, the binary tree data structure is used to define ordered binary trees. Section 5 shows how enumerated datatypes can be defined as simple forms of PVS datatypes. Section 6 shows the definition for disjoint unions. Mutually recursive datatypes are described in Section 7. Subtyping on recursive datatypes is described in Section 8. In Section 9, datatypes are used to construct effective representations for recursive ordinals which are then used as lexicographic termination measures for recursive functions. Section 10 shows some proofs about ordered binary trees which use some of the built-in simplifications shown in 11 along with the proof strategies described in Section 12. Some limitations of the PVS datatype mechanism are described in Section 13, followed by a discussion of related work in Section 14.

2 Lists: A Simple Abstract Datatype

The PVS prelude contains the following declaration of the abstract datatype of lists of a given element type.

```
                                                                        1
list[t:TYPE] : DATATYPE
 BEGIN
  null: null?
  cons (car: t, cdr :list) :cons?
 END list
```

Here `list` is declared as a type that is parametric in the type `t` with two *constructors* `null` and `cons`. The constructor `null` takes no arguments. The predicate *recognizer* `null?` holds for exactly those elements of the `list` datatype that are identical to `null`. The constructor `cons` takes two arguments where the first is of the type `t` and the second is a `list`. The recognizer predicate `cons?` holds for exactly those elements of the `list` type that are constructed using `cons`, namely, those that are not identical to `null`. There are two *accessors* corresponding to the two arguments of `cons`. The accessors `car` and `cdr` can be applied only to lists satisfying the `cons?` predicate so that `car(cons(x, l))` is `x` and `cdr(cons(x, l))` is `l`. Note that `car(null)` is not a well typed expression in that it generates a invalid proof obligation, a *type correctness condition* (TCC), that `cons?(null)` must hold.

The rules on datatype declarations as enforced by the PVS typechecker are:

1. The constructors must be pairwise distinct, i.e., there should be no duplication among the constructors.

2. The recognizers must be pairwise distinct, and also distinct from any of the constructors and the datatype name itself.

3. There must be at least one non-recursive constructor, that is, one that has no recursive occurrences of the datatype in its accessor types.[4]

[4]This is a needless restriction which will be removed in future versions of PVS. It was intended to ensure that the recursive datatype had a base object. However, it turns out that the restriction does not always guarantee the existence of such a base object such as when the base constructor has an accessor of an empty type. Also datatypes violating this restriction can be well-formed such as a datatype `okay` with one recursive constructor `mk_okay` that has one accessor `get` of type `list[okay]`. The base object in this case is `mk_okay(null)`. When there is no base object, then the datatype is empty.

4. The recursive occurrences of the datatype name in its definition must be positive as described in Section 2.1.

When the `list` abstract datatype is typechecked, three theories are generated in the file `list_adt.pvs`. The first theory, `list_adt`, contains the basic declarations and axioms formalizing the datatype, including an induction scheme and an extensionality axiom for each constructor. The second theory, `list_adt_map`, defines a `map` operation that lifts a function of type `[s -> t]` to a function of type `[list[s] -> list[t]]`. The third theory, `list_adt_reduce`, formalizes a general-purpose recursion operator over the abstract datatype. These theories are examined in more detail below for the case of binary trees. An important point to note about the generated datatype axioms is that apart from the induction and extensionality axioms, all the other axioms are automatically applied by PVS proof commands such as `assert` and `beta` so that the relevant axioms need never be explicitly invoked during a proof.

2.1 Positive type occurrence.

For each recursive datatype defined by means of the PVS `DATATYPE` declaration, the typechecker generates theories, definitions, and axioms similar to those shown above for the case of binary trees. In general, such a datatype can take individual and type parameters, and is specified in terms of the constructors, and the corresponding recognizers and accessors. The type of the accessor fields can be given recursively in terms of the datatype itself as long as this recursive occurrence of the type is *positive* in a certain restricted sense. A type occurrence `T` is positive in a type expression τ iff either

1. $\tau \equiv$ `T`.
2. `T` occurs positively in a supertype τ' of τ.
3. $\tau \equiv [\tau_1 \rightarrow \tau_2]$ where `T` occurs positively in τ_2. For example, `T` occurs positively in `sequence[T]` where `sequence[T]` is defined in the PVS prelude as the function type `[nat -> T]`.
4. $\tau \equiv [\tau_1, \ldots, \tau_n]$ where `T` occurs positively in some τ_i.
5. $\tau \equiv [\# \; l_1 \colon \tau_1, \ldots, l_n \colon \tau_n \; \#]$ where `T` occurs positively in some τ_i.
6. $\tau \equiv \mathit{datatype}[\tau_1, \ldots, \tau_n]$, where *datatype* is a previously defined datatype and `T` occurs positively in τ_i, where τ_i is a *positive parameter* of *datatype*.

The recursive occurrences of the datatype name in its definition must be positive so that we can assign a set-theoretic interpretation to all types. It is easy to see that violating this condition in the recursion leads to contradictions. For example, a datatype `T` with an accessor of type `[T -> bool]` would yield a contradiction since the cardinality of `[T -> bool]` is that of the power-set of `T` which by Cantor's theorem must be strictly greater than the cardinality of `T`. However, we have that distinct accessor elements lead to distinct datatype elements as well, and hence a contradiction. Similarly, an accessor type of `[[T -> bool] -> bool]` is also easily ruled out by cardinality considerations even though the occurrence of `T` in it is positive in terms of its polarity.

A *positive type parameter* `T` in a datatype declaration is one that only occurs positively in the type of an accessor. Positive type parameters in datatypes have a special role. As an example of a nested recursive datatype with recursion on the positive parameters, a *search tree* with leaf nodes bearing values of type `T` can be declared as in $\boxed{2}$. Note that the recursive occurrence of `leaftree` is as a (positive) parameter to the `list` datatype.

$\boxed{2}$

```
leaftree[T : TYPE] : DATATYPE
 BEGIN
  leaf(val : T) : leaf?
  node(subs : list[leaftree]): node?
 END leaftree
```

Positive datatype parameters are also used to generate the combinators `every`, `some`, and `map` which are described in detail for the datatype of binary trees in Section 3.

3 Binary Trees

A *binary tree* is a recursive data structure that in the base case is a *leaf* node, and in the recursive case consists of a *value* component, and *left* and *right* subtrees that are themselves binary trees. Binary trees can be formalized in several ways. In most imperative programming languages, they are defined as record structures containing pointers to the subtrees. They can also be encoded in terms of more primitive recursive data structures such as the s-expressions of Lisp. In a declarative specification language, one can formalize binary trees by enumerating the relevant axioms. One difficulty with this latter approach is the amount of effort involved in correctly identifying all of the relevant axioms. Another difficulty is that it can be tedious to explicitly invoke these axioms during a proof. This is the motivation for providing a concise abstract datatype mechanism in PVS that is integrated with the theorem prover. With binary trees, the declaration of the datatype is similar to that for lists above.

```
                                                                    3
binary_tree[T : TYPE] : DATATYPE
BEGIN
  leaf : leaf?
  node(val : T, left : binary_tree, right : binary_tree) : node?
END binary_tree
```

The two constructors `leaf` and `node` have corresponding recognizers `leaf?` and `node?`. The `leaf` constructor does not have any accessors. The `node` constructor has three arguments: the value at the node, the left subtree, and the right subtree. The accessor functions corresponding to these three arguments are `val`, `left`, and `right`, respectively. When the above datatype declaration is typechecked, the theories `binary_tree_adt`, `binary_tree_adt_map`, and `binary_tree_adt_reduce` are generated. The first of these has the form:

```
                                                                    4
binary_tree_adt[T: TYPE]: THEORY
  BEGIN

  binary_tree: TYPE

  leaf?, node?: [binary_tree -> boolean]

  leaf: (leaf?)

  node: [[T, binary_tree, binary_tree] -> (node?)]

  val: [(node?) -> T]

  left: [(node?) -> binary_tree]

  right: [(node?) -> binary_tree]

   { Various axioms and definitions omitted. }

  END binary_tree_adt
```

Note that the theory is parametric in the value type `T`. The first declaration above declares `binary_tree` as a type. The two recognizer predicates on binary trees `leaf?` and `node?` are then declared. The constructor `leaf` is declared to have type `(leaf?)` which is the subtype of `binary_tree` constrained by the `leaf?` predicate. The `node` constructor is declared as a function with domain type `[T, binary_tree, binary_tree]` and range type `(node?)` which is again the subtype of `binary_tree` constrained by the `node?` predicate. The three accessors on value (nonleaf) nodes are then declared. Each of these accessors takes as its domain the subset of binary trees that are constructed by means of the `node` constructor. Note that when `binary_tree_adt` is instantiated with an empty actual parameter type, the subtype `(node?)` must be empty since there is no value component corresponding to an element of `(node?)`.

The remainder of this section presents the axioms and definitions that are generated from the datatype declaration for binary trees. These axioms and definitions are not meant to be minimal and some of them are in fact redundant.

Definition by cases. The primitive `CASES` construct is used for definition by cases on the outermost constructor of a a PVS datatype expression. The syntax of the `CASES` construct is

`CASES` *expression* `OF` *selections* `ENDCASES`

where each *selection* (typically one selection per constructor) is of the form *pattern : expression* and a *pattern* is a constructor of arity n applied to n distinct variables. There are no explicit axioms characterizing the behavior of `CASES`. In the case of the binary tree datatype, when `w`, `x`, `y`, and `z` range over binary trees, `a` and `b` range over the parameter type `T`, `u` ranges over the range type `range`, and `v` ranges over the type `[T, binary_tree, binary_tree -> range]`, we implicitly assume the two axioms:

$$\begin{aligned} \texttt{CASES leaf OF leaf : u, node(a, y, z) : v(a, y, z)} &= \texttt{u} \\ \texttt{CASES node(b, w, x) OF leaf : u, node(a, y, z) : v(a, y, z)} &= \texttt{v(b, w, x)} \end{aligned}$$

Note that in the above axioms, the left-hand side occurrences of `a`, `y`, and `z` in `v(a, y, z)` are bound.

The `ord` function. The function `ord` assigns a number to a datatype construction, i.e., a datatype term given solely in terms of the constructors, according to its outermost constructor. The `ord` function is mainly used to enumerate the elements of an *enumerated type* (see Section 5). The `ord` function is defined using `CASES` in $\boxed{5}$.

5
```
ord(x: binary_tree): upto(1) =
  CASES x OF leaf: 0, node(node1_var, node2_var, node3_var): 1 ENDCASES
```

Thus `ord(leaf)` is `0`, whereas `ord(node(x, A, B))` is `1`.

Extensionality axioms. An extensionality axiom is generated corresponding to each constructor. The one for the `leaf` terms essentially asserts that `leaf` is the unique term of type `(leaf?)`.

```
                                                                6
binary_tree_leaf_extensionality: AXIOM
      (FORALL (leaf?_var: (leaf?), leaf?_var2: (leaf?)):
         leaf?_var = leaf?_var2);
```

For the `node` constructor, the extensionality axiom is:

```
                                                                7
binary_tree_node_extensionality: AXIOM
  (FORALL (node?_var: (node?)),
          (node?_var2: (node?)):
     val(node?_var) = val(node?_var2)
       AND left(node?_var) = left(node?_var2)
         AND right(node?_var) = right(node?_var2)
       IMPLIES node?_var = node?_var2)
```

In other words, any two value nodes that agree on all the accessors are equal.

Accessor–constructor axioms. Each accessor–constructor pair generates an axiom indicating the effect of applying the accessor to an expression constructed using the constructor. For example, the axiom corresponding to `val` and `node` has the form:

```
                                                                8
binary_tree_val_node: AXIOM
  (FORALL (node1_var: T), (node2_var: binary_tree),
          (node3_var: binary_tree):
     val(node(node1_var, node2_var, node3_var)) = node1_var)
```

We do not need an explicit axiom asserting that the recognizers `leaf?` and `node?` hold of disjoint subsets of the type of binary trees. This property can be derived from the `ord` function and the semantics of the `CASES` construct described above.

Eta axiom. The `eta` rule is a useful corollary to the extensionality axiom above and the accessor–constructor axioms shown above. It is introduced as an axiom in the `binary_tree_adt` theory as shown below though it does follow as a lemma from extensionality.[5]

[5]In future versions of PVS, it is intended that these will become lemmas with automatically generated proofs.

```
                                                                    9
binary_tree_node_eta: AXIOM
  (FORALL (node?_var: (node?)):
     node(val(node?_var), left(node?_var), right(node?_var)) = node?_var)
```

Structural induction. The theory `binary_tree_adt` also contains a structural induction scheme and a few recursion schemes. The induction scheme for binary trees is stated as:

```
                                                                    10
binary_tree_induction: AXIOM
  (FORALL (p: [binary_tree -> boolean]):
     p(leaf)
       AND
       (FORALL (node1_var: T), (node2_var: binary_tree),
               (node3_var: binary_tree): p(node2_var) AND p(node3_var)
            IMPLIES p(node(node1_var, node2_var, node3_var)))
       IMPLIES (FORALL (binary_tree_var: binary_tree): p(binary_tree_var)))
```

In other words, to prove a property of all binary trees, it is sufficient to prove in the base case that the property holds of the binary tree `leaf`, and that in the induction case, the property holds of a binary tree `node(v, A, B)` assuming (the induction hypothesis) that it holds of the subtrees `A` and `B`. One simple consequence of the induction axiom is the property that all binary trees are either leaf nodes or value nodes. This is also introduced as an axiom in the theory `binary_tree_adt`.

```
                                                                    11
binary_tree_inclusive: AXIOM
  (FORALL (binary_tree_var: binary_tree):
     leaf?(binary_tree_var) OR node?(binary_tree_var))
```

Definition by recursion. As another consequence of induction, we can demonstrate the existence and uniqueness of functions defined by structural recursion over binary trees. It is, however, convenient to have a more direct means for defining such recursive functions. PVS therefore provides various *recursion combinators*[6] which can be used to define recursive functions over datatype elements. One difficulty with defining a fully general recursion combinator is that it has to be parametric in the range type of the function being defined. Since PVS only provides such type parametricity at the level of theories, the generic recursion

[6] A combinator is a lambda expression without any free variables, but the term can also be applied to an operation that can be used as a building block for other operations.

combinators are defined in a separate theory `binary_tree_adt_reduce` which provides the additional type parameter. The recursion combinators for the common cases of functions returning natural numbers and sub-ϵ_0 ordinals (see Section 9) are defined in the theory `binary_tree_adt` itself.

The recursion combinator used for defining recursive functions over binary trees that return natural number values, is shown below. The idea is that we want to define a function f by the following recursion over binary trees:

$$\begin{aligned} f(\texttt{leaf}) &= a \\ f(\texttt{node}(\texttt{v},\texttt{A},\texttt{B})) &= g(\texttt{v}, f(\texttt{A}), f(\texttt{B})) \end{aligned}$$

We define such an f by taking a and g as arguments to the function `reduce_nat`. Note the use of the `CASES` construct to define a pattern-matching case split over a datatype value that in this case is a binary tree.

$\boxed{12}$

```
reduce_nat(leaf?_fun: nat, node?_fun: [[T, nat, nat] -> nat]):
    [binary_tree -> nat] =
  LAMBDA (binary_tree_adtvar: binary_tree):
    CASES binary_tree_adtvar OF
      leaf: leaf?_fun,
      node(node1_var, node2_var, node3_var):
          node?_fun(node1_var,
                    reduce_nat(leaf?_fun,
                               node?_fun)
                      (node2_var),
                    reduce_nat(leaf?_fun,
                               node?_fun)
                      (node3_var))
      ENDCASES;
```

The `reduce_nat` recursion combinator is useful for defining a "size" function as shown in $\boxed{22}$ but has the weakness that `node?_fun` only has access to the `val` field of the node. The theory `binary_tree_adt` also contains a variant `REDUCE_nat` where the `leaf?_fun` is a function and the `node?_fun` function takes an additional argument. The definition is omitted here since a more generic version of this recursion combinator is described below.

A generic version of the structural recursion combinator on binary trees is defined in `binary_tree_adt_reduce` where the type `nat` in the definition of `reduce_nat` has been generalized to an arbitrary parameter type `range`.

```
                                                                     13
binary_tree_adt_reduce[T: TYPE, range: TYPE]: THEORY
  BEGIN

  IMPORTING binary_tree_adt[T]

  reduce(leaf?_fun: range, node?_fun: [[T, range, range] -> range]):
      [binary_tree[T] -> range] =
    LAMBDA (binary_tree_var: binary_tree[T]):
      CASES binary_tree_var OF
        leaf: leaf?_fun,
        node(node1_var, node2_var, node3_var):
            node?_fun(node1_var,
                      reduce(leaf?_fun,
                             node?_fun)(node2_var),
                      reduce(leaf?_fun,
                             node?_fun)(node3_var))
        ENDCASES

   14

  END binary_tree_adt_reduce
```

The theory binary_tree_adt_reduce also contains the more flexible recursion combinator REDUCE where the leaf?_fun and node?_fun functions take binary_tree_var as an argument.

```
                                                                     14
  REDUCE(leaf?_fun: [binary_tree[T] -> range], node?_fun:
         [[T, range, range, binary_tree[T]] -> range]):
      [binary_tree[T] -> range] =
    LAMBDA (binary_tree_var: binary_tree[T]):
      CASES binary_tree_var OF
        leaf: leaf?_fun(binary_tree_var),
        node(node1_var, node2_var, node3_var):
            node?_fun(node1_var,
                      REDUCE(leaf?_fun,
                             node?_fun)(node2_var),
                      REDUCE(leaf?_fun,
                             node?_fun)(node3_var),
                      binary_tree_var)
        ENDCASES
```

PVS 2 introduced certain extensions to the datatype mechanism that were absent in

PVS 1. These include a primitive subterm relation, the `some`, `every`, and `map` combinators, and recursion through parameters of previously defined datatypes.

Subterm relation. The primitive subterm relation is defined on datatype objects and checks whether one object occurs as a (not necessarily proper) subterm of another object. This relation is defined as `subterm`.

15
```
subterm(x: binary_tree, y: binary_tree): boolean =
  x = y
    OR CASES y OF
         leaf: FALSE,
         node(node1_var, node2_var, node3_var):
             subterm(x, node2_var) OR subterm(x, node3_var)
         ENDCASES
```

The proper subterm relation is defined by `<<`. The proper subterm relation is useful as a well-founded termination relation that can be given along with the measure for a recursively defined function.

16
```
<<(x: binary_tree, y: binary_tree): boolean =
  CASES y OF
    leaf: FALSE,
    node(node1_var, node2_var, node3_var):
        (x = node2_var OR x << node2_var)
          OR x = node3_var OR x << node3_var
    ENDCASES
```

Well-foundedness. The next axiom asserts that datatype objects are well-founded with respect to the proper subterm relation. The induction axiom `binary_tree_induction` can be derived as a consequence of the axiom `binary_tree_well_founded` and the well-founded induction lemma `wf_induction` in the prelude.

17
```
binary_tree_well_founded: AXIOM well_founded?[binary_tree](<<);
```

The `every` combinator. The PVS typechecker generates the combinators `every` and `some` corresponding to the positive parameters of a datatype. For example, `every` checks if all values of this parameter type in an instance of the datatype satisfy a given predicate on the parameter type. Furthermore, if all the type parameters of a datatype are positive, then a `map` combinator is also generated.

The `every` combinator in the theory `binary_tree_adt` takes a predicate `p` on the positive type parameter `T`, and checks that every occurrence of an object of the type parameter in a binary tree satisfies the predicate. The `binary_tree_adt` theory also contains a non-curried variant of `every` that is written as `every(p, a)` instead of `every(p)(a)`.

18

```
every(p: PRED[T])(a: binary_tree): boolean =
  CASES a OF
    leaf: TRUE,
    node(node1_var, node2_var, node3_var):
        p(node1_var)
          AND every(p)(node2_var) AND every(p)(node3_var)
    ENDCASES
```

The `some` combinator. The `some` combinator is the dual to `every` and checks that some occurrence of a value of type `T` in the binary tree satisfies the given predicate.[7]

19

```
some(p: PRED[T])(a: binary_tree): boolean =
  CASES a OF
    leaf: FALSE,
    node(node1_var, node2_var, node3_var):
        p(node1_var) OR some(p)(node2_var) OR some(p)(node3_var)
    ENDCASES
```

The `map` combinator. Finally, when all the type parameters of a datatype definition occur positively in the definition, as is the case with `binary_tree`, a theory `binary_tree_adt_map` is generated that defines the curried and non-curried versions of the `map` combinator. In addition to the parameter `T`, `binary_tree_adt_map` takes a range type parameter `T1`. The `map` combinator applies a function `f` from `T` to `T1` to every value of type `T` in a given `binary_tree[T]` to return a result of type `binary_tree[T1]`. We omit the definition of the non-curried variant of `map`.

[7]For operations like `some` and `every`, PVS allows a notational convenience where `(some! x: p(x))` is shorthand for `some(lambda x: p(x))`.

20

```
binary_tree_adt_map[T: TYPE, T1: TYPE]: THEORY
  BEGIN

  IMPORTING binary_tree_adt

  map(f: [T -> T1])(a: binary_tree[T]): binary_tree[T1] =
    CASES a OF
      leaf: leaf[T1],
      node(node1_var, node2_var, node3_var):
          node[T1](f(node1_var),
                   map(f)(node2_var), map(f)(node3_var))
      ENDCASES

  END binary_tree_adt_map
```

In summary, the datatype mechanism accepts parametric recursive type definitions in terms of constructors, accessors, and recognizers. The recursive occurrences of the datatype must be positive. The typechecker generates recognizer subtypes, accessor-constructor axioms, extensionality axioms, a structural induction scheme, a subterm ordering relation, and various recursion combinators. With respect to positively occurring type parameters, the typechecker generates the `some` and `every` combinators. When all type parameters are positive, the typechecker also generates a `map` combinator. We next examine the use of the above theories formalizing binary trees in the definition of *ordered* binary trees.

4 Ordered Binary Trees

In ordered binary trees, the values in the nodes are ordered relative to each other: the value at a node is no less than any of the values in the left subtree, and no greater than any of the values in the right subtree. Such a data structure has many obvious uses since the values are maintained in sorted form and the average time for looking up a value or inserting a new value is logarithmic in the number of nodes.

The PVS specification of ordered binary trees is given in the theory `obt` below. It is worth noting the use of theory parameters in this specification. The body of the theory `obt` has been elided from the specification displayed below.

```
obt [T : TYPE,  <= : (total_order?[T])] : THEORY                           21
BEGIN
IMPORTING binary_tree[T]

A, B, C: VAR binary_tree
x, y, z: VAR T
pp: VAR pred[T]
i, j, k: VAR nat

 {definitions and lemmas shown below in 22 to 28}

END obt
```

The theory `obt` takes the type `T` of the values kept in the binary tree as its first parameter. Its second parameter is the total ordering used to order the binary tree. This parameter, represented as `<=`, has the type `(total_order?[T])` consisting of those binary relations on `T` that are total orderings, that is, those that are reflexive, transitive, antisymmetric, and linear. Note that the type of the second parameter to this theory depends on the first parameter `T`.

We can now use the `every` combinator to define when a binary tree is ordered relative to the theory parameter `<=`. This notion is captured by the predicate `ordered?` on binary trees. Since `ordered?` will be defined by a direct recursion, its definition will need a measure that demonstrates the termination of the recursion. In the definition of `size` below, the recursion combinator `reduce_nat` is used to count the number of value nodes in a given binary tree. This function is defined to return `0` when given a `leaf`, and to increment the sum of the sizes of the left and right subtrees by `1` when given a `node`.

```
                                                                           22
size(A) : nat =
  reduce_nat(0, (LAMBDA x, i, j: i + j + 1))(A)
```

The recursive definition of `ordered?` shown below returns `TRUE` in the base case since a leaf node is clearly an ordered tree by itself. In the recursive case, the definition ensures that the left and right subtrees of the given tree `A` are themselves ordered. It also uses `every` to check that all the values in the left subtree are no greater than the value `val(A)` at `A`, and the values in the right subtree are no less than the value at `A`. The measure `size` is used to demonstrate the termination of the recursion displayed by `ordered?`. The proper subterm relation shown in 16 could also be used as a well-founded relation in establishing the termination of `ordered?` by writing `MEASURE A BY <<` (see 34) in place of `MEASURE size`.

```
                                                              23
ordered?(A) : RECURSIVE bool =
  (IF node?(A) THEN (every((LAMBDA y: y<=val(A)), left(A)) AND
                     every((LAMBDA y: val(A)<=y), right(A)) AND
                     ordered?(left(A)) AND
                     ordered?(right(A)))
         ELSE TRUE ENDIF)
  MEASURE size
```

When the above definition is typechecked, two proof obligations (TCCs) are generated corresponding to the termination requirements for the two recursive calls. The first one requires that the size of the left subtree of a binary tree `A` must be smaller than the size of `A`. The second proof obligation requires that the size of the right subtree of `A` must be smaller than the size of `A`. Note how the governing `IF-THEN-ELSE` condition and the preceding conjuncts are included as antecedents in the proof obligations below.

```
                                                              24
ordered?_TCC1: OBLIGATION
      (FORALL (A):
         node?(A)
             AND every((LAMBDA y: y <= val(A)), left(A))
               AND every((LAMBDA y: val(A) <= y), right(A))
             IMPLIES size(left(A)) < size(A));

ordered?_TCC2: OBLIGATION
      (FORALL (v: [binary_tree[T] -> bool], A):
         node?(A)
             AND every((LAMBDA y: y <= val(A)), left(A))
               AND every((LAMBDA y: val(A) <= y), right(A)) AND v(left(A))
             IMPLIES size(right(A)) < size(A));
```

The PVS Emacs command `M-x tc` typechecks a file in PVS. The PVS Emacs command `M-x tcp` can be used to both typecheck the file and attempt to prove the resulting TCCs using the existing proof (if there is one) or a built-in strategy according to the source of the TCC (subtype, termination, existence, assuming, etc.). As it turns out, the `termination-tcc` strategy automatically proves both `ordered?_TCC1` and `ordered?_TCC2`.

The next definition in the `obt` theory is that of the `insert` operation. The term `insert(x, A)` returns that binary tree obtained by inserting the value `x` at the appropriate position in the binary tree `A`. The `insert` operation is also defined by recursion but employs the `CASES` construct instead of the `IF-THEN-ELSE` conditional. In the base case, when the argument `A` matches the term `leaf`, the binary tree containing the single value `x` is returned as the result. In the recursion case, the argument `A` has the form `node(y, B,` C), and if x is at most y according to the given total ordering on the type T, then we reconstruct the node with value y, left subtree insert(x, B), and right subtree C. Otherwise, we reconstruct the node with value y, left subtree B, and right subtree insert(x, C).

25
```
insert(x, A): RECURSIVE binary_tree[T] =
  (CASES A OF
    leaf: node(x, leaf, leaf),
    node(y, B, C): (IF x<=y THEN node(y, insert(x, B), C)
                            ELSE node(y, B, insert(x, C))
                    ENDIF)
   ENDCASES)
  MEASURE size(A)
```

When the above definition is typechecked, two termination proof obligations are generated corresponding to the two recursive invocations of insert. Both proof obligations insert_TCC1 and insert_TCC2 are automatically discharged by the default termination-tcc strategy.

26
```
insert_TCC1: OBLIGATION
      (FORALL (B: binary_tree[T], C: binary_tree[T], y: T, A, x):
         A = node(y, B, C) AND x <= y IMPLIES size(B) < size(A));

insert_TCC2: OBLIGATION
      (FORALL (B: binary_tree[T], C: binary_tree[T], y: T, A, x):
         A = node(y, B, C) AND NOT x <= y IMPLIES size(C) < size(A))
```

The following lemma states an interesting property of insert. Its proof requires the use of induction over binary trees. It asserts that if every value in the tree A has property pp, and the value x also has property pp, then every value in the result of inserting x into A has property pp.

27
```
ordered?_insert_step: LEMMA
   pp(x) AND every(pp, A) IMPLIES every(pp, insert(x, A))
```

The theorem ordered?_insert asserts the important property of insert that it returns an ordered binary tree when given an ordered binary tree.

28
```
ordered?_insert: THEOREM
   ordered?(A) IMPLIES ordered?(insert(x, A))
```

We examine some proofs of this theorem in Section 10.

5 In-line and Enumeration Types

The example of binary trees illustrated how abstract datatypes can be declared as theories (that are automatically expanded) within PVS. Abstract datatypes can be declared within other theories as long as they do not employ any parameters. Note that PVS has type parameterization only at the theory level and not at the declaration level. For example, the type of combinators constructed out of the K and S combinators is captured by the following declaration that can occur at the declaration level within a theory. The axioms generated by the `DATATYPE` declaration can be viewed using the PVS Emacs command `M-x ppe`.

29

```
combinators  : THEORY
  BEGIN
  combinators: DATATYPE
       BEGIN
         K: K?
         S: S?
         app(operator, operand: combinators): app?
       END combinators

  x, y, z: VAR combinators

  reduces_to: PRED[[combinators, combinators]]

  K: AXIOM reduces_to(app(app(K, x), y), x)
  S: AXIOM reduces_to(app(app(app(S, x), y), z), app(app(x, z), app(y, z)))

  END combinators
```

The most frequently used such *in-line* abstract datatypes are enumeration types. For example, the type of `colors` consisting of `red`, `white`, and `blue` can given by the following in-line datatype declaration.

30

```
  colors: DATATYPE
     BEGIN
       red: red?
       white: white?
       blue: blue?
     END colors
```

The above declaration is a rather verbose way of defining the type of `colors`. PVS provides an abbreviation mechanism that allows the above declaration to be expressed more succinctly as shown below.

```
                                                              31
  colors: TYPE = {red, white, blue}
```

All of the axiomatized properties of such enumeration types are built into the PVS proof checker as shown in the previous section so that no axioms about enumeration types need ever be explicitly used.

6 Disjoint Unions

The type constructor for the disjoint union of two types is popular enough to be included in several languages. The disjoint union of two sets A and B is a set in which each element is tagged according to whether it is from A or from B. It is easy to see that the type analogue of the disjoint union operation can be defined using the `DATATYPE` mechanism of PVS as shown below:

```
                                                              32
disj_union[A, B: TYPE] : DATATYPE
  BEGIN
    inl(left : A): inl?
    inr(right : B): inr?
  END disj_union
```

The type `disj_union[nat, bool]` then includes values such as `inl(1)` and `inr(TRUE)`.

Rushby [Rus95] presents a toy compiler verification exercise [WW93] in PVS and presents an extensive discussion of the use of disjoint unions in PVS specifications and proofs.

7 Mutually Recursive Datatypes

Mutually recursive datatypes arise quite frequently in programming and specification. A common example is that of a language definition where type expressions contain terms and vice-versa. Mutually recursive type definitions are not directly admissible using the PVS datatype mechanism. But most typical mutual recursive types can, in fact, be defined as a single datatype in PVS with subtypes that group together classes of constructors. PVS 2 has been extended to admit such datatypes with *sub-datatypes*. The example below describes the class of arithmetic expressions that include numbers, sums, and conditional expressions classified by the sub-datatype `term`, where the test component of a conditional expression is a boolean expression classified by the subdatatype `expr`. Thus sub-datatypes are a way of collecting together groups of constructors of a datatype that form one part of a mutually recursive datatype definition. In the example below, boolean expressions are defined as equalities between arithmetic expressions, and conditional arithmetic expressions contain boolean subexpressions, so that arithmetic and boolean expressions are mutually recursive.

33

```
arith: DATATYPE WITH SUBTYPES expr, term
 BEGIN
  num(n:int): num?              :term
  sum(t1:term,t2:term):   sum?  :term
% ...
  eq(t1: term, t2: term): eq?   :expr
  ift(e: expr, t1: term, t2: term): ift? :term
% ...
 END arith
```

The only restriction on the use of subdatatypes other than those listed in Section 2 is that the sub-datatypes should be pairwise distinct and differ from the datatype itself. In particular, sub-datatypes need not actually be used in which case they are empty. It is possible to define mutual recursive types that lead to empty constructor subtypes such as if the `eq` constructor in the `arith` datatype was specified as `eq(t1: expr, t2: expr): eq? : expr`.

An evaluator for such arithmetic/boolean expressions can be defined as `eval` whose range type is a disjoint union of `bool` and `int` (according to whether the input expression is of type `expr` or `term`. The function `eval` is therefore dependently typed to return values of type `(bool?)` on inputs of type `expr` and values of type `(int?)` on inputs of type `term`.

```
                                                                        34
arith_eval: THEORY
 BEGIN
  IMPORTING arith

  value: DATATYPE
   BEGIN
    bool(b:bool): bool?
    int(i:int): int?
   END value

  eval(a: arith): RECURSIVE
         {v: value | IF expr(a) THEN bool?(v) ELSE int?(v) ENDIF} =
   CASES a OF
    num(n):       int(n),
    sum(n1, n2):  int(i(eval(n1)) + i(eval(n2))),
    eq(n1, n2):   bool(i(eval(n1)) = i(eval(n2))),
    ift(e, n1, n2): IF b(eval(e)) THEN eval(n1) ELSE eval(n2) ENDIF
   ENDCASES
   MEASURE a BY <<

 END arith_eval
```

8 Lifting Subtyping on Recursive Datatype Parameters

The datatype mechanism in PVS 2.0 had the limitation that though the type of `nat` of natural numbers is a subtype of the type `int` of integers, the type `list[nat]` of lists over the natural numbers is not a subtype of the type `list[int]` of lists over the integers. The datatype mechanism in PVS 2.1 has been modified to lift such subtyping over positive parameters to the corresponding abstract datatypes. In general, given a datatype `D` with a positive type parameter, we have

`D[{x: T | p(x)}]` $\equiv$ `{d: D[T] | every(p)(d)}`.

While `cons[nat]` is neither syntactically nor semantically identical to `cons[int]`, constructor *applications* involving `cons[int]` and `cons[nat]` such as `cons[nat](0, null[nat])` and `cons[int](0, null[int])` are syntactically identical. Also, constructors that are declared to have no accessors (e.g., `null`) are syntactically equal, so `null[int]` $\equiv$ `null[real]`, but `null[int]` and `null[bool]` belong to incompatible types.

In general, when a constructor, accessor, or recognizer occurs as an operator of an application, the actual parameter is only used for testing compatibility. Note that the actual parameter is not actually ignored. For example, the expression `cons[nat](-1, null)` is not type correct and generates the unprovable proof obligation `-1 > 0`.

When multiple parameters are involved, only the positive ones satisfy the subtyping equivalences given above. Thus in the datatype declaration

```
dt[t1, t2: TYPE, c: t1]: DATATYPE
 BEGIN
  b: b?
  c(a1:[t1 -> t2], a2: dt): c?
 END dt
```

only `t2` occurs positively, so `dt[int, nat, 3]` is a subtype of `dt[int, int, 3]`, but bears no relation to `dt[nat, nat, 3]` or to `dt[int, nat, 2]`.

9 Representations of Recursive Ordinals

Ordinals are needed to provide lexicographic termination measures for recursive functions. The Ackermann function provides a well known example of a doubly recursive function that requires a lexicographic measure. Péter's version [Pét67] of the Ackermann function is defined in the theory `ackermann` as `ack`.

35

```
ackermann: THEORY
BEGIN
i, j, k, m, n: VAR nat

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 lex2(m, n)

  ⋮
END ackermann
```

The lexicographic termination measure for `ack` is computed by the function `lex2` (see 39) which returns a representation for the ordinal in the lexicographic ordering. The ordinal ϵ_0 is the least ordinal x such that $x = \omega^x$, and therefore includes $0, 1, \ldots, \omega, \omega + 1, \ldots \omega +$

$\omega, \ldots, 3 * \omega, \ldots, \omega^2, \ldots, \omega^\omega, \ldots, \omega^{\omega^{\cdot^{\cdot^{\omega}}}}, \ldots$. The sub-$\epsilon_0$ ordinals can be represented using the Cantor normal form which asserts that to any non-zero ordinal α, there are n ordinals $\alpha_1, \ldots, \alpha_n$ with $\alpha_1 \leq \ldots \leq \alpha_n < \alpha$, such that $\alpha = \omega^{\alpha_1} + \omega^{\alpha_2} + \ldots + \omega^{\alpha_n}$. We can make this representation slightly more compact by adding natural number coefficients so that to any α, there are ordinals $\alpha_1, \ldots, \alpha_n$ such that $\alpha_1 \leq \ldots \leq \alpha_n < \alpha$, and natural numbers $c_1, \ldots, c_n$ such that $\alpha = c_1 * \omega^{\alpha_1} + c_2 * \omega^{\alpha_2} + \ldots + c_n * \omega^{\alpha_n}$. It is easy to see that a lexicographic measure can be given by $n * \omega^0 + m * \omega$ which is just $n + m * \omega$.

We now explain how the sub-ϵ_0 ordinals are defined in the PVS prelude. We start by defining an `ordstruct` datatype that represents ordinal-like structures.

36

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

In intuitive terms, the ordinal represented by `zero` is `0`, and the ordinal represented by `add(c, alpha, beta)` given by, say *ordinal*(`add(c, alpha, beta)`) is $\texttt{c} * \omega^{\mathit{ordinal}(\texttt{alpha})} + \mathit{ordinal}(\texttt{beta})$. We can then define an ordering relation on `ordstruct` terms as given by `<` in $\boxed{37}$. It compares `add(i, u, v)` against `add(j, z, w)` by either recursively ensuring `u < z`, or checking that `u` is syntactically identical to `z` and either `i < j` or `i = j` and recursively `v < w`.

37

```
ordinals: THEORY
 BEGIN
  i, j, k: VAR posnat
  m, n, o: VAR nat
  u, v, w, x, y, z: VAR ordstruct
  size: [ordstruct->nat] = reduce[nat](0, (LAMBDA i, m, n: 1 + m+n));

  <(x, y): RECURSIVE bool =
     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 size(x);
```

This is not quite the ordering relation we want since it will obviously only work for normalized (and therefore, canonical) representations where the exponent ordinals appear in sorted (decreasing) order. In particular, note that the use of syntactic identity on `ordstruct` terms will not work unless the terms are in fact canonical representatives. It is easy to define a predicate which identifies an `ordstruct` term as being in the required Cantor normal form by defining a recursive predicate `ordinal?` as shown in $\boxed{38}$.

38

```
>(x, y): bool = y < x;
<=(x, y): bool = x < y OR x = y;
>=(x, y): bool = y < x OR y = x

ordinal?(x): RECURSIVE bool =
  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: NONEMPTY_TYPE = (ordinal?)
```

The definition of `ordinal?` checks `add(i, u, v)` to recursively ensure that `u` and `v` are ordinals, and that in `add(i, u, add(k, r, s))`, we have `r < u`. This latter use of the ordering relation is acceptable since we have already checked that `r` and `u` are proper normal forms. The definition of `lex2` is given in $\boxed{39}$. Note that `add(n, zero, zero)` represents n, `add(m, add(1, zero, zero), zero)` represents $m * \omega$, and `add(m, add(1,zero,zero), add(n,zero, zero))` represents $n + m * \omega$.[8]

[8] The PVS `CONVERSION` mechanism can be used to gracefully embed the natural numbers into the `ordinal` type by converting `0` to `zero`, and a positive number `n` to `add(n, zero, zero)`.

39

```
lex2(m, n): ordinal =
  (IF m=0
     THEN IF n = 0
             THEN zero
             ELSE add(n, zero, zero)
          ENDIF
     ELSIF n = 0 THEN add(m, add(1,zero,zero),zero)
     ELSE add(m, add(1,zero,zero), add(n,zero, zero))
   ENDIF)

lex2_lt: LEMMA
  lex2(i, j) < lex2(m, n) =
   (i < m OR (i = m AND j < n))
```

Returning to the example of the Ackermann function in [35], the measure `lex2(m, n)` generates three termination TCCs corresponding to the three recursive invocations of the function.

40

```
ack_TCC2: OBLIGATION
      (FORALL (m, n): NOT m = 0 AND n = 0
        IMPLIES lex2(m - 1, 1) < lex2(m, n));

ack_TCC5: OBLIGATION
      (FORALL (m, n):
         NOT m = 0 AND NOT n = 0
          IMPLIES lex2(m, n - 1) < lex2(m, n));

ack_TCC6: OBLIGATION
      (FORALL (v: [[nat, naturalnumber] -> nat], m, n):
         NOT m = 0 AND NOT n = 0
             IMPLIES lex2(m - 1, v(m, n - 1)) < lex2(m, n));
```

All three TCCs are proved automatically by the default `termination-tcc` strategy.

10 Some Illustrative Proofs about Ordered Binary Trees

We present two proofs of `ordered?_insert` shown in [28]. The second proof exhibits a greater level of automation than the first proof. The first proof illustrates the various low-level datatype related proof commands that are provided by PVS, and the second proof illustrates how these commands can be combined to form more powerful and automatic proof strategies. Strategies are similar to the *tactics* of the LCF [GMW79] family of proof checkers.

10.1 A Low-level Proof

When we invoke `M-x pr` on `ordered?_insert`, the theorem to be proved is displayed in the `*pvs*` buffer, and we are prompted for an inference rule by the `Rule?` prompt. Since the proof is by induction, the first step in the proof is the command `(induct "A")`. This indicates that we wish to invoke the `induct` strategy with `A` as the induction variable. The induction strategy finds the induction axiom corresponding to the datatype of `A`, instantiates it suitably, and simplifies it to generate the base and induction cases. We are then presented the base case of the proof. (The induction case can be displayed with the PVS Emacs command `M-x siblings`.

```
                                                                    41
 ordered?_insert :
  |-------
{1}   (FORALL (A: binary_tree[T], x: T):
         ordered?(A) IMPLIES ordered?(insert(x, A)))
Rule? (induct "A")
Inducting on A,
this yields  2 subgoals:
ordered?_insert.1 :
  |-------
{1}   (FORALL (x: T): ordered?(leaf) IMPLIES ordered?(insert(x, leaf)))
```

In the next step, we replace the universally quantified variable with a Skolem constant and flatten the sequent by simplifying all top-level propositional connectives that are disjunctive (*i.e.*, negations, positive implications and disjunctions, and negative conjunctions).

```
                                                                    42
Rule? (skosimp)
Skolemizing and flattening,
this simplifies to:
ordered?_insert.1 :
{-1}   ordered?(leaf)
  |-------
{1}   ordered?(insert(x!1, leaf))
```

The obvious step now is to open up the definitions of `insert` and `ordered?`. This is done by two invocations of the `expand` rule.

```
                                                              43
Rule? (expand "insert")
Expanding the definition of insert,
this simplifies to:
ordered?_insert.1 :
[-1]   ordered?(leaf)
  |-------
{1}   ordered?(node(x!1, leaf, leaf))

Rule? (expand "ordered?")
Expanding the definition of ordered?,
this simplifies to:
ordered?_insert.1 :
  |-------
{1}   (every((LAMBDA (y: T): y <= x!1), leaf)
           AND every((LAMBDA (y: T): x!1 <= y), leaf)
             AND ordered?(leaf) AND ordered?(leaf))
```

The problem now is that all the occurrences of `ordered?` are expanded so that the antecedent formula `ordered?(leaf)` reduces to `TRUE` and vanishes from the sequent. This formula in its unexpanded form is actually useful since it occurs in the consequent part of the sequent. We could press on and expand `ordered?` once again or, alternatively, we could undo this step of the proof and expand `ordered?` more selectively using the command `(expand "ordered?" +)`.

```
                                                                   44
Rule? (undo)
This will undo the proof to:
ordered?_insert.1 :
[-1]   ordered?(leaf)
  |-------
{1}   ordered?(node(x!1, leaf, leaf))
Sure? (Y or N): y
ordered?_insert.1 :
[-1]   ordered?(leaf)
  |-------
{1}   ordered?(node(x!1, leaf, leaf))
Rule? (expand "ordered?" +)
Expanding the definition of ordered?,
this simplifies to:
ordered?_insert.1 :
[-1]   ordered?(leaf)
  |-------
{1}   (every((LAMBDA (y: T): y <= x!1), leaf)
          AND every((LAMBDA (y: T): x!1 <= y), leaf)
            AND ordered?(leaf) AND ordered?(leaf))
```

Now an invocation of `assert` eliminates the occurrences of the subformula `ordered?(leaf)` in the consequent since it appears in the antecedent. Expanding `every` then completes the base case of the proof without any further work.

```
                                                                   45
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
ordered?_insert.1 :
[-1]   ordered?(leaf)
  |-------
{1}   (every((LAMBDA (y: T): y <= x!1), leaf)
          AND every((LAMBDA (y: T): x!1 <= y), leaf))
Rule? (expand "every")
Expanding the definition of every,
this simplifies to:
ordered?_insert.1 :
[-1]   ordered?(leaf)
  |-------
{1}   TRUE
which is trivially true.
This completes the proof of ordered?_insert.1.
```

Having completed the base case of the proof, we are left with the induction case. Our first step here is to apply the rule `skosimp*`. This is a strategy that repeatedly performs a `skolem!` followed by a `flatten` until nothing changes, i.e., it is an iterated form of the `skosimp` command.

```
                                                                          46
ordered?_insert.2 :
  |-------
{1}   (FORALL (node1_var: T, node2_var: binary_tree[T],
               node3_var: binary_tree[T]):
         (FORALL (x: T):
            ordered?(node2_var) IMPLIES ordered?(insert(x, node2_var)))
             AND
           (FORALL (x: T):
              ordered?(node3_var) IMPLIES ordered?(insert(x, node3_var)))
             IMPLIES
           (FORALL (x: T):
              ordered?(node(node1_var, node2_var, node3_var))
                  IMPLIES
                ordered?(insert(x, node(node1_var, node2_var, node3_var)))))
Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
ordered?_insert.2 :
{-1}   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
{-2}   (FORALL (x: T):
         ordered?(node3_var!1) IMPLIES ordered?(insert(x, node3_var!1)))
{-3}   ordered?(node(node1_var!1, node2_var!1, node3_var!1))
  |-------
{1}   ordered?(insert(x!1, node(node1_var!1, node2_var!1, node3_var!1)))
```

Now we have a subgoal sequent in which the induction hypotheses are the formulas number `-1` and `-2`, and the induction conclusion formulas are numbered `-3` and `1`. We clearly need to expand the definitions of `insert` and `ordered?` in the induction conclusion. We first expand `insert` and then propositionally simplify the resulting `IF-THEN-ELSE` expression as shown below.

```
                                                                          47
Rule? (expand "insert" +)
Expanding the definition of insert,
this simplifies to:
ordered?_insert.2 :
[-1]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-2]   (FORALL (x: T):
         ordered?(node3_var!1) IMPLIES ordered?(insert(x, node3_var!1)))
[-3]   ordered?(node(node1_var!1, node2_var!1, node3_var!1))
  |-------
{1}   IF x!1 <= node1_var!1
        THEN ordered?(node(node1_var!1, insert(x!1, node2_var!1), node3_var!1))
      ELSE ordered?(node(node1_var!1, node2_var!1, insert(x!1, node3_var!1)))
      ENDIF

Rule? (prop)
Applying propositional simplification,
this yields  2 subgoals:
ordered?_insert.2.1 :
{-1}   x!1 <= node1_var!1
[-2]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-3]   (FORALL (x: T):
         ordered?(node3_var!1) IMPLIES ordered?(insert(x, node3_var!1)))
[-4]   ordered?(node(node1_var!1, node2_var!1, node3_var!1))
  |-------
{1}   ordered?(node(node1_var!1, insert(x!1, node2_var!1), node3_var!1))
```

The propositional simplification step generates two subgoals according to whether the recursive invocation of `insert` is on the left or the right subtree. We first consider the insertion into the left subtree given by subgoal `ordered?_insert.2.1`. We can instantiate the induction hypothesis numbered `-2` by applying the `inst?` command which uses syntactic matching to find instantiating terms for the universally quantified variable in `-2`.

```
                                                                  48
Rule? (inst?)
Found substitution:
x gets x!1,
Instantiating quantified variables,
this simplifies to:
ordered?_insert.2.1 :
[-1]   x!1 <= node1_var!1
{-2}   ordered?(node2_var!1) IMPLIES ordered?(insert(x!1, node2_var!1))
[-3]   (FORALL (x: T):
         ordered?(node3_var!1) IMPLIES ordered?(insert(x, node3_var!1)))
[-4]   ordered?(node(node1_var!1, node2_var!1, node3_var!1))
  |-------
[1]   ordered?(node(node1_var!1, insert(x!1, node2_var!1), node3_var!1))
```

The next step is to expand the definition of `ordered?` in the induction conclusion. Note that the second argument to the `expand` proof command is a list of the formula numbers where the expansion is to be performed. It makes the proof considerably less robust if it explicitly mentions such formula numbers, though this can be unavoidable in some cases.[9]

```
                                                                  49
Rule? (expand "ordered?" (-4 1))
Expanding the definition of ordered?,
this simplifies to:
ordered?_insert.2.1 :
[-1]   x!1 <= node1_var!1
[-2]   ordered?(node2_var!1) IMPLIES ordered?(insert(x!1, node2_var!1))
[-3]   (FORALL (x: T):
         ordered?(node3_var!1) IMPLIES ordered?(insert(x, node3_var!1)))
{-4}   (every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
           AND every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
             AND ordered?(node2_var!1) AND ordered?(node3_var!1))
  |-------
{1}   (every((LAMBDA (y: T): y <= node1_var!1), insert(x!1, node2_var!1))
           AND every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
             AND ordered?(insert(x!1, node2_var!1)) AND ordered?(node3_var!1))
```

Applying propositional simplification `prop` to the resulting subgoal generates two further subgoals. The first of these is easily proved by rewriting using the lemma `ordered?_insert_step`. Note that this is a conditional rewrite rule and has the form $A \supset B$, where the rewriting given by B can be applied to a matching instance $\sigma(B)$ only

[9]PVS is currently being enhanced to allow labels to be introduced for sequent formulas so that formula selection in the PVS proof commands can be done with labels as an alternative to formula numbers.

when the corresponding $\sigma(A)$ (the condition) is provable. The `rewrite` proof strategy attempts to discharge these conditions automatically, and any undischarged conditions are generated as subgoals.

```
                                                                        50
Rule? (prop)
Applying propositional simplification,
this simplifies to:
ordered?_insert.2.1 :
{-1}   ordered?(insert(x!1, node2_var!1))
[-2]   x!1 <= node1_var!1
[-3]   (FORALL (x: T):
         ordered?(node3_var!1) IMPLIES ordered?(insert(x, node3_var!1)))
{-4}   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
{-5}   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
{-6}   ordered?(node2_var!1)
{-7}   ordered?(node3_var!1)
  |-------
{1}   every((LAMBDA (y: T): y <= node1_var!1), insert(x!1, node2_var!1))
Rule? (rewrite "ordered?_insert_step")
Found matching substitution:
A gets node2_var!1,
x gets x!1,
pp gets (LAMBDA (y: T): y <= node1_var!1),
Rewriting using ordered?_insert_step,
This completes the proof of ordered?_insert.2.1.
```

We have now completed the part of the proof corresponding to the insertion into the left subtree. Next, we proceed to the case when the `insert` operation is applied to the right subtree. This case is similar to the proof of `ordered?_insert.2.1`.

```
                                                                        51
ordered?_insert.2.2 :
[-1]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-2]   (FORALL (x: T):
         ordered?(node3_var!1) IMPLIES ordered?(insert(x, node3_var!1)))
[-3]   ordered?(node(node1_var!1, node2_var!1, node3_var!1))
  |-------
{1}   x!1 <= node1_var!1
{2}   ordered?(node(node1_var!1, node2_var!1, insert(x!1, node3_var!1)))
```

As in $\boxed{48}$ earlier, we apply the step `inst?`.

```
                                                                          52
Rule? (inst?)
Found substitution:
x gets x!1,
Instantiating quantified variables,
this simplifies to:
ordered?_insert.2.2 :
{-1}   ordered?(node2_var!1) IMPLIES ordered?(insert(x!1, node2_var!1))
[-2]   (FORALL (x: T):
         ordered?(node3_var!1) IMPLIES ordered?(insert(x, node3_var!1)))
[-3]   ordered?(node(node1_var!1, node2_var!1, node3_var!1))
  |-------
[1]   x!1 <= node1_var!1
[2]   ordered?(node(node1_var!1, node2_var!1, insert(x!1, node3_var!1)))
```

It however instantiates the formula `-1` which is not the appropriate induction hypothesis for the right branch. To apply the `inst?` step with greater selectivity, we undo the last step and supply a further argument to `inst?` indicating the number of the quantified formula to be instantiated.

```
                                                                          53
Rule? (inst? -2)
Found substitution:
x gets x!1,
Instantiating quantified variables,
this simplifies to:
ordered?_insert.2.2 :
[-1]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
{-2}   ordered?(node3_var!1) IMPLIES ordered?(insert(x!1, node3_var!1))
[-3]   ordered?(node(node1_var!1, node2_var!1, node3_var!1))
  |-------
[1]   x!1 <= node1_var!1
[2]   ordered?(node(node1_var!1, node2_var!1, insert(x!1, node3_var!1)))
```

Now, as before, we expand the definition of `ordered?` in the induction conclusion formulas `-3` and `2`.

```
                                                                          54
Rule? (expand "ordered?" (-3 2))
Expanding the definition of ordered?,
this simplifies to:
ordered?_insert.2.2 :
[-1]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-2]   ordered?(node3_var!1) IMPLIES ordered?(insert(x!1, node3_var!1))
{-3}   (every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
           AND every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
             AND ordered?(node2_var!1) AND ordered?(node3_var!1))
  |-------
[1]   x!1 <= node1_var!1
{2}   (every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
           AND
         every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
             AND ordered?(node2_var!1) AND ordered?(insert(x!1, node3_var!1)))
```

Propositional simplification yields a single goal sequent.

```
                                                                          55
Rule? (prop)
Applying propositional simplification,
this simplifies to:
ordered?_insert.2.2 :
{-1}   ordered?(insert(x!1, node3_var!1))
[-2]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
{-3}   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
{-4}   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
{-5}   ordered?(node2_var!1)
{-6}   ordered?(node3_var!1)
  |-------
{1}   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
[2]   x!1 <= node1_var!1
```

As before, we attempt to rewrite the formula 1 using the lemma `ordered?_insert_step`, but as shown in $\boxed{56}$, this does not terminate the current branch of the proof.

```
                                                                    56
Rule? (rewrite "ordered?_insert_step")
Found matching substitution:
A gets node3_var!1,
x gets x!1,
pp gets (LAMBDA (y: T): node1_var!1 <= y),
Rewriting using ordered?_insert_step,
this simplifies to:
ordered?_insert.2.2 :
[-1]   ordered?(insert(x!1, node3_var!1))
[-2]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-3]   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
[-4]   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
[-5]   ordered?(node2_var!1)
[-6]   ordered?(node3_var!1)
  |-------
{1}   node1_var!1 <= x!1
[2]   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
[3]   x!1 <= node1_var!1
```

We are left with having to discharge one of the conditions of the rewrite rule, namely `node1_var!1 <= x!1`. This follows from the other consequent formula `x!1 <= node1_var!1` and the observation that `<=` here is a linear ordering. The proof now requires that the type information of `<=` be made explicit using the `typepred` command.

```
                                                                    57
Rule? (typepred "<=")
<= does not uniquely resolve - one of:
  obt.<= : (total_order?[T]),
  reals.<= : [[real, real] -> bool],
  ordinals.<= : [[ordstruct, ordstruct] -> bool]
Restoring the state.
ordered?_insert.2.2 :
[-1]   ordered?(insert(x!1, node3_var!1))
[-2]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-3]   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
[-4]   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
[-5]   ordered?(node2_var!1)
[-6]   ordered?(node3_var!1)
  |-------
{1}   node1_var!1 <= x!1
[2]   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
[3]   x!1 <= node1_var!1
```

However, the command (typepred "<=") does not succeed since the typechecker is unable to resolve among the many possible references for <=. The more explicit command (typepred "obt.<=") does succeed.[10]

58

```
Rule? (typepred "obt.<=")
Adding type constraints for  obt.<=,
this simplifies to:
ordered?_insert.2.2 :
{-1}   total_order?[T](obt.<=)
[-2]   ordered?(insert(x!1, node3_var!1))
[-3]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-4]   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
[-5]   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
[-6]   ordered?(node2_var!1)
[-7]   ordered?(node3_var!1)
  |-------
[1]   node1_var!1 <= x!1
[2]   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
[3]   x!1 <= node1_var!1
```

We then expand the definition of total_order?.

59

```
Rule? (expand "total_order?")
Expanding the definition of total_order?,
this simplifies to:
ordered?_insert.2.2 :
{-1}   partial_order?(obt.<=) & dichotomous?(obt.<=)
[-2]   ordered?(insert(x!1, node3_var!1))
[-3]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-4]   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
[-5]   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
[-6]   ordered?(node2_var!1)
[-7]   ordered?(node3_var!1)
  |-------
[1]   node1_var!1 <= x!1
[2]   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
[3]   x!1 <= node1_var!1
```

[10] Note that in PVS 2.1, the typechecking of input expressions to proof commands automatically resolves such ambiguities in favor of expressions occurring in the goal sequent. Thus, this ambiguity is no longer reported.

Applying `flatten` removes the conjunction in `-1`.

```
                                                                      60
Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
ordered?_insert.2.2 :
{-1}   partial_order?(obt.<=)
{-2}   dichotomous?(obt.<=)
[-3]   ordered?(insert(x!1, node3_var!1))
[-4]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-5]   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
[-6]   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
[-7]   ordered?(node2_var!1)
[-8]   ordered?(node3_var!1)
  |-------
[1]   node1_var!1 <= x!1
[2]   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
[3]   x!1 <= node1_var!1
```

Expanding the definition of `dichotomous?` yields the needed linearity property of the ordering relation.

```
                                                                      61
Rule? (expand "dichotomous?")
Expanding the definition of dichotomous?,
this simplifies to:
ordered?_insert.2.2 :
[-1]   partial_order?(obt.<=)
{-2}   (FORALL (x: T), (y: T): (obt.<=(x, y) OR obt.<=(y, x)))
[-3]   ordered?(insert(x!1, node3_var!1))
[-4]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-5]   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
[-6]   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
[-7]   ordered?(node2_var!1)
[-8]   ordered?(node3_var!1)
  |-------
[1]   node1_var!1 <= x!1
[2]   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
[3]   x!1 <= node1_var!1
```

When this linearity property is heuristically instantiated, we get a tautologous subgoal that is polished off with `prop`, thus completing the proof.

```
                                                                    62
Rule? (inst?)
Found substitution:
y gets x!1,
x gets node1_var!1,
Instantiating quantified variables,
this simplifies to:
ordered?_insert.2.2 :
[-1]   partial_order?(obt.<=)
{-2}   (obt.<=(node1_var!1, x!1) OR obt.<=(x!1, node1_var!1))
[-3]   ordered?(insert(x!1, node3_var!1))
[-4]   (FORALL (x: T):
         ordered?(node2_var!1) IMPLIES ordered?(insert(x, node2_var!1)))
[-5]   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
[-6]   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
[-7]   ordered?(node2_var!1)
[-8]   ordered?(node3_var!1)
  |-------
[1]   node1_var!1 <= x!1
[2]   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
[3]   x!1 <= node1_var!1
Rule? (prop)
Applying propositional simplification,
This completes the proof of ordered?_insert.2.2.
This completes the proof of ordered?_insert.2.
Q.E.D.
Run time  = 12.32 secs.
Real time = 1916.88 secs.
```

The above exercise illustrates several aspects of PVS proofs of theorems involving abstract datatypes. The `induct` strategy automatically employs the datatype induction scheme. Most of the datatype axioms need never be explicitly invoked in a proof — the above proof does not mention any datatype axioms.

More general lessons about PVS are also illustrated by the above exercise. Primary among these are the use of `undo` to backtrack in a proof, the use of `expand` and `rewrite` to rewrite using definitions and rewrite rules, `assert` to simplify using the decision procedures and the assertions in the sequent, and `inst?` to heuristically instantiate a suitably quantified variable using matching.

We now examine a more automated proof of the same theorem.

10.2 A Semi-automated Proof

We can now retry the proof of the theorem `ordered?_insert` using a more high-level strategy defined in PVS. This strategy is called `induct-and-simplify`. It applies the `induct` strategy and then tries to complete the proof by repeatedly skolemizing and instantiating quantifiers, and applying the decision procedures, rewrite rules, and propositional simplification. We employ as rewrite rules, the lemma `ordered?_insert_step` and any definitions used directly or indirectly in the statement of the theorem. The script shown below has been automatically generated from the `induct-and-simplify` command up to the subgoal in 66. The first segment of the proof shows the setting up of the rewrite rules mentioned in the `induct-and-simplify` command.

63

```
 ordered?_insert :
  |-------
{1}   (FORALL (A: binary_tree[T], x: T):
         ordered?(A) IMPLIES ordered?(insert(x, A)))
Rule? (induct-and-simplify "A" :rewrites "ordered?_insert_step")
```

The internal steps of the strategy are not displayed but any applications of rewrite rules are indicated in the proof commentary. This rewriting commentary can be turned off using the proof command `(rewrite-msg-off)` or controlled using the PVS Emacs command `M-x set-rewrite-depth`. The rewriting in the base case is shown below in 64.

64

```
ordered? rewrites ordered?(leaf)
  to TRUE
insert rewrites insert(x!1, leaf)
  to node(x!1, leaf, leaf)
every rewrites every((LAMBDA (y: T): y <= x!1), leaf)
  to TRUE
every rewrites every((LAMBDA (y: T): x!1 <= y), leaf)
  to TRUE
ordered? rewrites ordered?(node(x!1, leaf, leaf))
  to TRUE
```

The rewriting steps occurring in the induction case are shown in 65

```
                                                                        65
ordered? rewrites ordered?(node(node1_var!1, node2_var!1, node3_var!1))
  to (every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
           AND every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
             AND ordered?(node2_var!1) AND ordered?(node3_var!1))
insert rewrites insert(x!1, node(node1_var!1, node2_var!1, node3_var!1))
  to (IF x!1 <= node1_var!1
         THEN node(node1_var!1, insert(x!1, node2_var!1), node3_var!1)
       ELSE node(node1_var!1, node2_var!1, insert(x!1, node3_var!1))
       ENDIF)
ordered? rewrites
  ordered?(node(node1_var!1, insert(x!1, node2_var!1), node3_var!1))
  to (every((LAMBDA (y: T): y <= node1_var!1), insert(x!1, node2_var!1))
           AND every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
             AND ordered?(insert(x!1, node2_var!1)) AND ordered?(node3_var!1))
ordered? rewrites
  ordered?(node(node1_var!1, node2_var!1, insert(x!1, node3_var!1)))
  to (every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
           AND
         every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
             AND ordered?(node2_var!1) AND ordered?(insert(x!1, node3_var!1)))
ordered?_insert_step rewrites
  every((LAMBDA (y: T): y <= node1_var!1), insert(x!1, node2_var!1))
  to TRUE
```

One subgoal results from the `induct-and-simplify` command as shown in 66. This subgoal is nearly the same as subgoal `ordered?_insert.2.2` in 55 from the previous proof attempt. This means that the `induct-and-simplify` strategy completed the base case and most of the induction branch of the proof automatically. The subgoal in 66 corresponds to the case of insertion into the right subtree. The strategy failed to complete this branch of the proof because it was unable to apply the rewrite rule `ordered?_insert_step` automatically. This application failed because one of the conditions of the rewrite rule, `node1_var!1 <= x!1`, could not be discharged. This condition follows from formula number `1` in 66 and the linearity of the `<=` relation. The latter constraint is, however, buried in the type constraint `(total_order?)` of `<=`. This information has to be made explicit before the proof can be successfully completed.

```
                                                                    66
By induction on A, and by repeatedly rewriting and simplifying,
this simplifies to:
ordered?_insert :
{-1}   ordered?(insert(node1_var!1, node2_var!1))
{-2}   ordered?(insert(x!1, node3_var!1))
{-3}   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
{-4}   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
{-5}   ordered?(node2_var!1)
{-6}   ordered?(node3_var!1)
  |-------
{1}   x!1 <= node1_var!1
{2}   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
```

The rest of proof can be completed interactively as in the previous proof attempt but we attempt a slightly different sequence of steps this time. The first step is identical to that in $\boxed{56}$ where the `ordered?_insert_step` lemma is manually invoked as a rewrite rule using the `rewrite` strategy.

```
                                                                    67
Rule? (rewrite "ordered?_insert_step")
Found matching substitution:
A gets node3_var!1,
x gets x!1,
pp gets (LAMBDA (y: T): node1_var!1 <= y),
Rewriting using ordered?_insert_step,
this simplifies to:
ordered?_insert :
[-1]   ordered?(insert(node1_var!1, node2_var!1))
[-2]   ordered?(insert(x!1, node3_var!1))
[-3]   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
[-4]   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
[-5]   ordered?(node2_var!1)
[-6]   ordered?(node3_var!1)
  |-------
{1}   node1_var!1 <= x!1
[2]   x!1 <= node1_var!1
[3]   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
```

The next step is also identical to that of $\boxed{58}$ where the type constraints for the `<=` operator are explicitly introduced into the sequent.

68

```
Rule? (typepred "obt.<=")
Adding type constraints for  obt.<=,
this simplifies to:
ordered?_insert :
{-1}   total_order?[T](obt.<=)
[-2]   ordered?(insert(node1_var!1, node2_var!1))
[-3]   ordered?(insert(x!1, node3_var!1))
[-4]   every((LAMBDA (y: T): y <= node1_var!1), node2_var!1)
[-5]   every((LAMBDA (y: T): node1_var!1 <= y), node3_var!1)
[-6]   ordered?(node2_var!1)
[-7]   ordered?(node3_var!1)
  |-------
[1]   node1_var!1 <= x!1
[2]   x!1 <= node1_var!1
[3]   every((LAMBDA (y: T): node1_var!1 <= y), insert(x!1, node3_var!1))
```

The main difference from the previous proof attempt is that we now invoke a somewhat powerful variant of the all-purpose `grind` strategy where the `:if-match` flag is set to `all` indicating that all matching instances of any quantified formulas are to be used. If we do not supply this option, the heuristic instantiator picks the wrong instance since the type constraints for `<=` themselves provide matching instances for the one relevant type constraint, namely, `dichotomous?(obt.<=)`.

69

```
Rule? (grind :if-match all)
reflexive? rewrites reflexive?(obt.<=)
  to FORALL (x: T): obt.<=(x, x)
transitive? rewrites transitive?(obt.<=)
  to FORALL (x: T), (y: T), (z: T): obt.<=(x, y) & obt.<=(y, z) => obt.<=(x, z)
preorder? rewrites preorder?(obt.<=)
  to FORALL (x: T): obt.<=(x, x)
          & FORALL (x: T), (y: T), (z: T):
          obt.<=(x, y) & obt.<=(y, z) => obt.<=(x, z)
antisymmetric? rewrites antisymmetric?(obt.<=)
  to FORALL (x: T), (y: T): obt.<=(x, y) & obt.<=(y, x) => x = y
partial_order? rewrites partial_order?(obt.<=)
  to (FORALL (x: T): obt.<=(x, x)
           & FORALL (x: T), (y: T), (z: T):
           obt.<=(x, y) & obt.<=(y, z) => obt.<=(x, z))
          & FORALL (x: T), (y: T): obt.<=(x, y) & obt.<=(y, x) => x = y
dichotomous? rewrites dichotomous?(obt.<=)
  to (FORALL (x: T), (y: T): (obt.<=(x, y) OR obt.<=(y, x)))
total_order? rewrites total_order?[T](obt.<=)
  to ((FORALL (x: T): obt.<=(x, x)
            & FORALL (x: T), (y: T), (z: T):
            obt.<=(x, y) & obt.<=(y, z) => obt.<=(x, z))
           & FORALL (x: T), (y: T): obt.<=(x, y) & obt.<=(y, x) => x = y)
          & (FORALL (x: T), (y: T): (obt.<=(x, y) OR obt.<=(y, x)))
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
Run time  = 48.86 secs.
Real time = 230.49 secs.
```

The above semi-automated proof attempt illustrates the power that is gained from combining high-level strategies (e.g., `induct-and-simplify` and `grind`) to handle the easy portions of a proof with low-level manual interaction to carry out the more delicate steps. Note that the inner workings of these strategies which are hidden in the above proof can be observed by invoking them with a `$` suffix as in `induct-and-simplify$` and `grind$`.

The proofs of the lemmas `ordered?_insert_step` in 27 and `search_insert` shown in 70 below can be completed automatically by the single command:

```
(induct-and-simplify "A")
```

70

```
search(x, A): RECURSIVE bool =
  (CASES A OF
    leaf: FALSE,
    node(y, B, C): (IF x = y THEN TRUE
                      ELSIF x<=y THEN search(x, B)
                        ELSE search(x, C)
                    ENDIF)
   ENDCASES)
  MEASURE size(A)

search_insert: THEOREM search(y, insert(x, A)) = (x = y OR search(y, A))
```

10.3 Proof Status

To conclude the development of binary trees and ordered binary trees, we can apply the PVS Emacs command `M-x prt` to recheck all the proofs and print out the proof status.[11] The output of this command is shown below. It indicates that not only have all the theorems been proved but so have any non-axioms (lemmas, TCCs, etc.) used in any of these proofs.

71

```
Proof summary for theory obt
    ordered?_TCC1.........................................proved - complete
    ordered?_TCC2.........................................proved - complete
    insert_TCC1...........................................proved - complete
    insert_TCC2...........................................proved - complete
    ordered?_insert_step..................................proved - complete
    ordered?_insert.......................................proved - complete
    search_insert.........................................proved - complete
    Theory totals: 7 formulas, 7 attempted, 7 succeeded.
```

11 Built-in Datatype Simplifications

As indicated at the outset, the primary advantage of using abstract datatypes in PVS is that a lot of knowledge about such datatypes and their operations is built into the system. To illustrate the sort of datatype simplifications that are built into PVS, consider the theory `binary_props` shown below.

[11] The command `M-x status-proof-theory` or `M-x spt` can be used to get the proof status without rechecking the proofs.

72

```
binary_props[T : TYPE] : THEORY

  BEGIN
  IMPORTING binary_tree_adt[T]
  A, B, C, D: VAR binary_tree[T]
  x, y, z: VAR T

  leaf_leaf : LEMMA leaf?(leaf)
  node_node : LEMMA node?(node(x, B, C))

  leaf_leaf1: LEMMA A = leaf IMPLIES leaf?(A)

  node_node1: LEMMA A = node(x, B, C) IMPLIES node?(A)

  val_node: LEMMA val(node(x, B, C)) = x

  leaf_node: LEMMA NOT (leaf?(A) AND node?(A))

  node_leaf: LEMMA leaf?(A) OR node?(A)

  leaf_ext: LEMMA (FORALL (A, B: (leaf?)): A = B)

  node_ext:  LEMMA (FORALL (A : (node?)) :
                   node(val(A), left(A), right(A)) = A)

  END binary_props
```

All the lemmas excluding the last one, `node_ext`, are provable by the command `(then (skosimp)(assert))`. This means that the `assert` rule builds in several simplifications. In the case of `leaf_leaf` and `node_node`, `assert` can reduce the application of a recognizer to a constructor expression to either `TRUE` or `FALSE`. In the case of `leaf_leaf1` and `node_node1`, it even can do this simplification across an equality. The reason for this simplification is that subtype information is asserted to the decision procedures so that when `A = node(x, B, C)` is asserted to the decision procedures, so is `node?(node(x, B, C))`, and `node?(A)` is deduced by congruence closure in the decision procedures. The simplifications in `leaf_leaf` and `node_node`, but not `leaf_leaf1` and `node_node1`, can also be carried out by the PVS beta-reduction rule `beta` since this rule does not make use of equality information.

The lemma `val_node` illustrates that the application of an accessor to a constructor expression yields the appropriate field of the constructor expression. This simplification can also be done by the `beta` rule.

The simplification implicit in `leaf_node` is more subtle and captures the exclusivity property of abstract datatypes. Here, from an antecedent formula `leaf?(A)`, `assert` is able to simplify the expression `node?(A)` to `FALSE` since no datatype expression can satisfy two recognizers. The simplification implicit in `node_leaf` captures the inclusivity property of abstract datatypes. Here, `assert` is able to determine that a recognizer holds of an expression by demonstrating that all the other recognizers are false on the expression. In general, when confronted with the application of a recognizer r to a datatype expression e, the simplifier evaluates the truth value of each recognizer of that datatype when applied to the given expression using the decision procedures. If $r(e)$ is determined to be `TRUE` by the decision procedures, then $r(e)$ is obviously simplified to `TRUE` by the simplifier. If for some other recognizer r', $r'(e)$ is determined to be `TRUE` by the decision procedures, then $r(e)$ is simplified to `FALSE`. If for all recognizers r' distinct from r, $r'(e)$ is determined to be `FALSE`, then $r(e)$ is simplified to `TRUE`.

The lemma `leaf_ext` essentially illustrates that for constructors such as `leaf` that have no accessors, there is no distinction between the forms `leaf?(A)` and `A = leaf`. It also illustrates how the subtype information is used implicitly to simplify `A = B` to `TRUE`.

The lemma `node_ext` is the only one that cannot be proved by the command `(then (skosimp)(assert))`. Here, this command simplifies the goal sequent to a single subgoal that is then proved by means of the `(apply-extensionality)` command. This illustrates that the extensionality axiom for datatypes is built into the primitive PVS rule `extensionality` and is also employed by the strategies `replace-extensionality` and `apply-extensionality`.

12 Some Proof Strategies

We briefly explain the definitions of the proof strategies `induct-and-simplify` and `grind` that were used in Section 10. The PVS manuals [OSR93] provides more details. These strategies are quite useful for proofs of datatype-related theorems. The `induct-and-simplify` strategy takes an argument list of the form:

```
(var &optional (fnum 1) name (defs t) (if-match best)
               theories rewrites exclude)
```

where

- `var` is the induction variable and is the only required argument
- `fnum` is the formula number of the induction formula where the induction variable is universally quantified; it defaults to `1`

- `name` names the induction scheme to be employed
- `defs` indicates which definitions of constants used in the current goal are to be installed as rewrite rules; it defaults to `t` indicating that all relevant definitions must be installed
- `if-match` instructs the heuristic instantiator to use none, one, all, or the best matching instantiation for a quantified formula; it defaults to `best`
- `theories` is the list of theories to be installed as rewrite rules
- `rewrites` is the list of formulas or definitions that are to be installed as rewrite rules, and
- `exclude` is a list of formulas or definitions that should be removed from the rewrite rule base

The body of the definition of the strategy has the form:

```
(then
 (install-rewrites$ :defs defs :theories theories :rewrites rewrites
  :exclude exclude)
 (try (induct var fnum name)
  (then (skosimp*) (assert) (repeat (lift-if))
   (repeat*
    (then (assert) (bddsimp) (skosimp*)
     (if if-match (inst? :if-match if-match) (skip)) (lift-if))))
  (skip)))
```

The `induct-and-simplify` strategy first installs the rewrites using `install-rewrites$` on `defs`, `theories`, `rewrites`, and `exclude`. It then introduces the appropriate instance of the induction scheme using `induct`. Then the strategy carries out one round of skolemization (introduction of new constants for outermost universally bound variables) using `skosimp*`, rewriting and simplification using `assert`, and repeated lifting of conditionals to the top level using `lift-if`. Following this, there are repeated rounds of rewriting/simplification, propositional simplification, skolemization, heuristic instantiation, and if-lifting until each resulting subgoal has stabilized.

The `grind` strategy is similar. It takes the following argument list:

```
(&optional (defs !) theories rewrites exclude (if-match t) (updates? t))
```

where the only new argument from `induct-and-simplify` is `updates?` which when set to `NIL` blocks the if-lifting of update applications of the form `(A WITH [(i) := b])(j)` to `(IF i = j THEN b ELSE A(j) ENDIF)`.

The body of the definition of `grind` is:

```
(then
 (install-rewrites$ :defs defs :theories theories :rewrites rewrites
  :exclude exclude)
 (then (bddsimp) (assert)) (replace*)
 (reduce$ :if-match if-match :updates? updates?))
```

Here the rewrite rules are installed using `install-rewrites$`, and followed by propositional simplification, rewriting and simplification, equality replacement, followed by repeated applications of these steps along with heuristic instantiation and if-lifting.

It should be clear from the above definition that it is fairly straightforward to write powerful proof strategies using the constructs provided by the PVS proof checker.

13 Limitations of the PVS Abstract Datatype Mechanism

The abstract datatype mechanism of PVS is intended to capture a fairly large class of datatypes whose axioms can be easily and systematically generated. This class contains all the freely generated term algebras over an order-sorted signature which includes the various stack and tree-like data structures. It excludes such important datatypes as integers (which are built into PVS), bags, sets, and queues. It also excludes various lazy data structures such as lazy lists or streams. These latter structures can be introduced by implementing a similar mechanism for introducing co-datatypes as for datatypes.

The `DATATYPE` mechanism is a primitive construct of PVS and is not merely a definitional extension of PVS. It therefore has the disadvantage that it is not possible to prove general theorems about all recursive datatypes in the way that one can about all inductive definitions given as least fixed points. For example, Bird's fusion theorem [Bir95] cannot be uniformly proved for all recursive datatypes and has to be proved for each datatype individually [Sha96].

14 Related Work

There are a number of algebraic specification languages such as Larch [GJMW93], OBJ [FGJM85], and ACT-ONE [EM85] that can be used to specify abstract datatypes but these specifications are manually axiomatized and not automatically generated from a succinct description as is the case with the PVS `DATATYPE` construct. The axioms are used as rewrite rules so that there is no built-in automation of the simplification of datatype expressions.

The programming language ML [MTH90] has a similar recursive datatype mechanism. Unlike the PVS mechanism, the ML construct allows arbitrary forms of recursion. As noted earlier, such recursive type definitions do not always have a proper set-theoretic semantics. Gunter [Gun93] explains how certain recursive datatypes that are admissible in ML can lead to unsoundnesses if admitted into a higher-order logic.

The HOL system has a mechanism for defining abstract datatypes [Mel89] that is somewhat more restrictive than that of PVS: there are more constraints on recursion and HOL lacks the useful notion of subtyping that is available in PVS. However, the HOL construct is definitional in that a recursively specified datatype is defined in terms of the primitive type constructors available in HOL. In particular, any newly defined recursive datatype is shown to be interpretable as a subset of some existing datatype based on finitely branching trees. The axioms generated from the datatype declaration are shown to be sound with respect to this interpretation. Isabelle/ZF and Isabelle/HOL both have a similar but more general facility for defining datatypes and co-datatypes [Pau97]. The Isabelle datatype mechanism also accomodates infinitely branching trees. The Coq system has a facility for defining recursive and co-recursive datatypes which, like PVS and unlike HOL and Isabelle, is a primitive construct of the Coq logic [Gim96].

The shell principle used in the Boyer-Moore theorem prover [BM79, BM88] is quite similar to the PVS `DATATYPE` mechanism. It permits recursive datatypes to be specified by means of constructors, accessors, and recognizers. Like PVS, the axioms corresponding to a shell datatype are built into the inference mechanisms of the theorem prover. The shell principle, however, has many serious limitations. It is complicated by the lack of types or subtypes in the Boyer-Moore logic. The shell principle only allows one constructor and a bottom object thus ruling out a great many useful datatypes where multiple constructors are required.

15 Conclusions

We have described the `DATATYPE` mechanism of PVS and demonstrated its use in proof construction. This mechanism captures a large class of commonly used type definitions within a succinct notation. A number of facts about these automatically generated abstract datatypes are built into the inference mechanisms of PVS so that it is possible to obtain a significant degree of automation when proving theorems involving datatypes. The high level of automation in the low-level inference mechanisms in PVS makes it easy to define powerful and flexible high-level proof strategies.

Acknowledgements. The design and implementation of PVS was directed by John Rushby of the SRI Computer Science Laboratory. He, along with Rick Butler of NASA and Mandayam Srivas, suggested several improvements to this document. Donald Syme of Cambridge University carefully proofread the document and gave numerous helpful suggestions. We are also grateful to Ulrich Hensel of TU Dresden for his as yet unheeded suggestion to incorporate corecursive datatypes.

Bibliography

[Bir95] Richard S. Bird. Functional algorithm design. In Bernhard Möller, editor, *Mathematics of Program Construction '95*, number 947 in Lecture Notes in Computer Science, pages 2–17. Springer Verlag, 1995.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, NY, 1979.

[BM88] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, NY, 1988.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, Berlin, 1985.

[FGJM85] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian K. Reid, editor, *12th ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[Gim96] Eduardo Giménez. A tutorial on recursive types in Coq. Draft. Available from `ftp://cri.ens-lyon.fr/pub/LIP/COQ/V6.1.beta/doc/RecTutorial.ps.gz`, March 1996.

[GJMW93] John V. Guttag and James J. Horning with S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer Verlag, 1993.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[Gun93] Elsa L. Gunter. Why we can't have SML style datatype declarations in HOL. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, number 20 in IFIP Transactions A, pages 561–568. North-Holland, 1993.

[HJ97] Ulrich Hensel and Bart Jacobs. Proof principles for datatypes with iterated recursion. In *Category Theory in Computer Science*, 1997. Also appears as Technical Report CSI-R9703, Computing Science Institute, Faculty of Mathematics and Informatics, Catholic University of Nijmegen.

[Mel89] Thomas F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Theorem Proving*, pages 341–386, New York, NY, 1989. Springer-Verlag.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[OSR93] 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. Three volumes: Language, System, and Prover Reference Manuals; A new edition for PVS Version 2 is expected in late 1996.

[Pau97] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7:175–204, March 1997.

[Pét67] R. Péter. *Recursive Functions.* Academic Press, New York, NY, 1967.

[Rus95] John Rushby. Proof Movie II: A proof with PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995. Available, with specification files, from `http://www.csl.sri.com/movie.html`.

[Sha96] N. Shankar. Steps towards mechanizing program transformations using PVS. *Science of Computer Programming*, 26(1–3):33–57, 1996.

[WW93] Debora Weber-Wulff. Proof Movie—a proof with the Boyer-Moore prover. *Formal Aspects of Computing*, 5(2):121–151, 1993.