A Verification System

Rasa Bonyadlou

8th August 2003

AALBORG UNIVERSITY
Department of Computer Science

TITLE: A Verification System

SEMESTER PERIOD:
SSE4,
2nd of February 2003- 8th of August 2003

PROJECT GROUP:
B1-207c

AUTHOR:
Rasa Bonyadlou, rasa2@cs.auc.dk

SUPERVISOR:
Anders P. Ravn, apr@cs.auc.dk

NUMBER OF PAGES: 98

TOTAL NUMBER OF COPIES: 5 and 1 online

APPENDIXES: PVS Grammar, CD ROM

Preface

This project is submitted as a final Master Thesis in Software System Engineering in the Department of Computer Science at Aalborg University, Denmark, within the Distributed System and Semantic unit, August 2003. The purpose of the project is to implement the PVS (Prototype Verification System). For the implementation of this project JavaCC and Jbuilder will be used. Enclosed within this project report is a CD ROM with the source code for PVS GUI, parser, compiler and type checker, which is implemented in Java, and several PVS theories.

Rasa Bonyadlou

Aalborg University, August 2003.

Abstract

The Prototyping Verification System (PVS) developed by Stanford Research Institute (SRI), is an interactive theorem prover for a typed higher order logic. PVS is a system for writing specifications and constructing proofs. PVS provides an integrated environment for the development and analysis of formal specifications, and supports a wide range of activities involved in creating, analyzing and documenting theories and proofs. The aim of this project is to describe the PVS system and re-implement it in Java.

Contents

List of Figures

Acknowledgements

I have the following people to thank, for without them, writing this report would have been impossible.
I am grateful to my supervisor Prof. Anders P. Ravn for his useful discussions and necessary suggestions on all parts of the project and implementation.
I am also thankful to Piotr Makowski for his useful assistance and valuable contributions for the first part of the project and implementation.

Chapter 1

Introduction

In a theorem proving approach to verification, a system and its properties are described by means of logical formula and the system is verified by means of a logical proof of the desired properties.

A popular support tool for the formal specification and verification activities is the theorem prover PVS or *Prototyping Verification System*. PVS supports both a highly expressive specification language and a very effective interactive theorem prover in which most of the low-level proof steps are automated, [2].
Although PVS provides some simple proof combinators, PVS is mainly used for development and analysis of formal specifications. The PVS system consists of a specification language, a parser for the language, a type checker, a prover, specification libraries, and various browsing tools. Its theorem prover, or proof checker, is both interactive and highly mechanized: the user chooses each step that is to be applied and PVS performs it, displays the results, and then waits for the next command. Furthermore, PVS is designed to help in the detection of errors and confirmation of correctness, by type checking. Many formal specifications contain significant errors when first written, and automated proof checking can be one of the *fastest* ways to detect errors. PVS is the most recent in the line of specification languages, theorem provers, and verification system, and it is implemented in Lisp. PVS is an interactive theorem prover, it is not automate theorem prover, because decidable logic often not rich enough, or inconvenient. Semi- automated theorem provers usually don't tell what is wrong, experience and knowledge about internals are required.

This project consists of two main parts, the aim of the first part of the project is to describe the initial steps towards the re-implementation of PVS, the main concepts of the theorem prover and logic system and development of the PVS parser. And the aim of the second part of the project is to develop PVS GUI and type checker. The major goals of this project are:

- Document the architecture of a verification system.
- Implement it in a portable language.
- Have an open source implementation of PVS parser, compiler, type checker and graphical user interface.

Application of PVS

PVS is mainly intended for the formalization of requirements and design-level specifications, and for the analysis of intricate and difficult problems. It (and its predecessors) have been applied to algorithms and architectures for fault-tolerant flight control systems, and to problems in hardware and real-time system design. Collaborative projects involving PVS are ongoing with NASA and several aerospace companies; applications include a microprocessor for aircraft flight-control, diagnosis and scheduling algorithms for fault-tolerant architectures, and requirements specification for portions of the Space Shuttle flight-control system.

PVS has been installed at hundreds of sites in North America, Europe, and Asia; current work is developing PVS methodologies for highly automated hardware verification (including integration with model checkers), and for concurrent and real-time systems (including a transparent embedding of the duration calculus).

PVS Work flow

Generally there are two main steps in PVS work flow, that can be used in PVS system.

- The system (programs, circuit, protocol,...) and their properties can be converted to the PVS system. Conversion can be automated or done manually.
- Proof constructions, which are interaction with the theorem prover.

Figure 1.1 is the work flow for PVS.

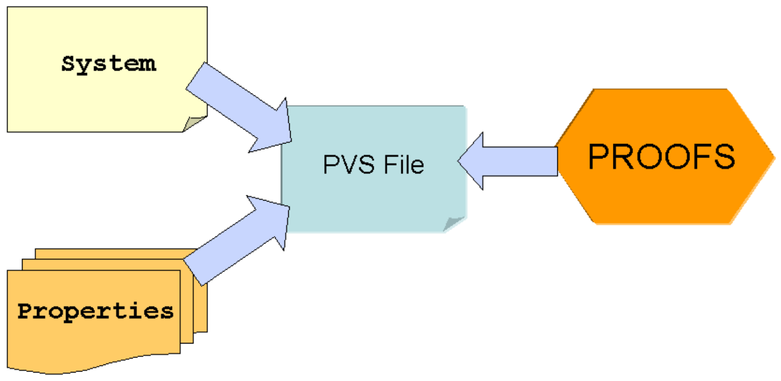


Figure 1.1: PVS Flow

Report Structure

The structure of the report includes eight main chapters. The following chapter introduces the PVS system architecture and shows the main parts of PVS and the abstract way of constructing PVS proofs, it also contains the main concepts of the PVS languages. The third chapter describes the PVS parser, lexical analysis and the lexical structure in PVS. The fourth chapter covers the main concepts of the logic and the theorem prover, it introduces the first order and propositional logic, binary decision trees and PVS tactics. The type checking chapter covers the PVS type system and PVS prelude file. The design chapter covers internal representation of PVS parser which will be implemented according to the PVS grammar in JavaCC, the GUI for PVS system and the internal representation of PVS type checker and theorem prover. The testing chapter contains the test specification, instructions and outputs for PVS parser, GUI and type checker. The summary of the project is in the conclusion chapter. Appendix A contains the whole grammar of PVS and appendix B contains the source code of the PVS parser, type checker and GUI, and some examples for testing. Mainly this project will discus and implement the PVS parser, type checker, GUI and the main concepts of the logic and theorem prover.

Chapter 2

Architecture Of PVS

This chapter introduce the PVS parts and the initial steps for proving a theorem in PVS. It will give basic concepts of the proofs and proof goals in PVS with an example proof. The PVS languages is discussed in the last part of the chapter.

PVS consists of a specification language, a number of predefined theories, a theorem prover, various utilities, documentation, and several examples that illustrate different methods of using the system in several application areas. The main parts of the PVS are: editor, parser, theorem prover with proof checking and type checker. Figure 2.1 shows use cases for PVS parts.

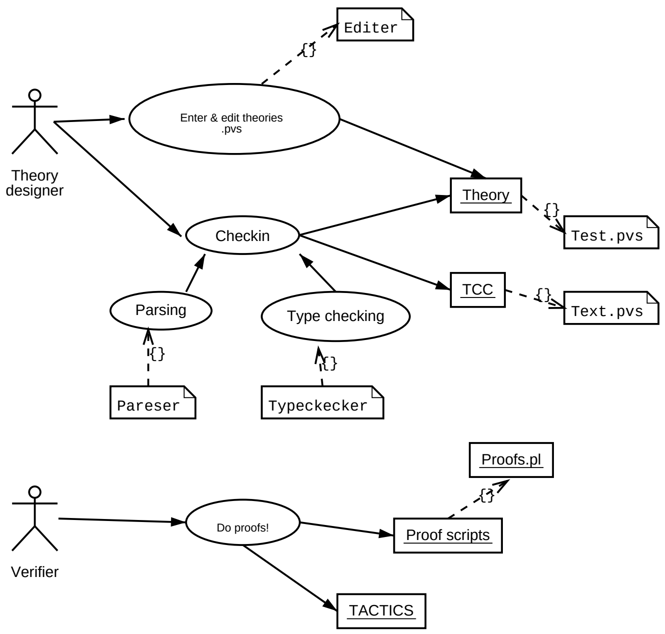


Figure 2.1: PVS-Architecture

The actors in the figure are:

- Theory Designer: This user specifies the theory or theories in the PVS specification language, where it is parsed and typed checked.
- Verifier: This user constructs a proof of the theorems. The proof is recorded in a proof script.

1 A Simple Example Of a PVS Specification

A PVS specification consists of a number of files, which contains some theories. Each theory takes a list of theory parameters and provides a list of declarations or definitions for variables, individual constants, types and formulas. All the specification files in PVS must have a *.pvs* extension. As specifications are developed, their proofs are kept in files of the same name with *.prf* extension. Consider a simple example specification:

```
sum: THEORY
 BEGIN
   n: VAR nat
   sum(n): RECURSIVE nat=
   (IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF)
   MEASURE (LAMBDA n:n)
   closed_form:THEOREM sum(n)= (n* (n+1)) / 2
  END sum
```

The user is supposed to enter the theory and start PVS to prove the theorem, (user guide for PVS specification [10]). Following are steps for starting PVS to prove the theorems:

1.1 Creating The Specification

The first step is, to enter the `sum` theory with an editor, e.g., the PVS emacs based editor and then save it with a `.pvs` extension. This simple theory has no parameters and contains three declarations. The first declares `n` to be a logical variable of type `nat`, the built-in type of natural numbers. The next declaration is a definition of the function `sum(n)`, whose value is the sum of the first `n` natural numbers. Associated with this definition is a `measure` function, following the `MEASURE` key word. A measure function is a function from the arguments of the recursive function into the natural numbers. In order to make the recursion well-formed one must prove that the measure is strictly decreasing for recursive calls. Here, the `MEASURE` is the identity function. The final declaration is a formula, a `THEOREM` which gives the closed form of the `sum`.

1.2 Parsing

The `sum` specification can be parsed and if there is any problem during parsing, the system will show an error message in an error window. Then, the user should fix the error and parse it once again. The parsing starts with a lexical analysis for all parts of the theory and it generates the parse tree for it. The details of the parsing will be discussed in chapter 3.

1.3 Type Checking

Type checking is a simple and effective way to discover errors in specifications. Type checking checks for semantic errors, such as undeclared names and ambiguous types. Type checking may build a new file or internal structure with *Type Correctness Condition* (TCC)s [9]. The user is expected to discharge these proof obligations with the PVS prover. Proof of TCCs can be postponed, but the system keeps track of all undischarged proof obligations and the affected theories and theorems are marked as incompletely proved. When the theory has been type checked, a message will display any TCCs generated.

2 Proving

The user supplies the steps in the argument and PVS applies them to the goal of the proof, breaking them into simpler subgoals or obvious truths. If all the subgoals are reduced to obvious truths, the proof attempt has succeeded. Otherwise, the proof attempt fails either because the argument or the conjecture is incorrect. The main property of the design assumptions in PVS are:

- The purpose of an automated proof checker is not only to prove theorems but also to provide useful feedback from failed and partial proofs.
- Automation should also be used to capture repetitive patterns of argumentation.
- The end product of a proof attempt should be a proof that, with only a small amount of work, can be made humanly readable so that it can be subjected to the process of mathematical scrutiny.

2.1 The Structure Of PVS Proofs

This section is based on the description in the PVS prover guide[7]. The prover maintains a *proof tree*, and the goal of the proofs is to construct a proof tree which is complete, in the sense that all of the leaves are recognized as true. The nodes of the proof tree are sequents, and the proving parts will always be looking for unproved branches. Each proof goal is a *sequent* consisting of a sequence of formulas called *antecedents* and a sequence of formulas called *consequents.* In PVS such a sequent is displayed as:

$$\begin{array}{cl} \{-1\} & A_1 \\ \{-2\} & A_2 \\ [-3] & A_3 \\ & \vdots \\ \texttt{|------} & \\ [1] & B_1 \\ \{2\} & B_2 \\ \{3\} & B_3 \\ & \vdots \end{array}$$

where the A_i and B_j are PVS formulas collectively referred to as *sequent formulas*: the A_i are the antecedents and the B_j are the consequents; the row of dashes separates the antecedents from the consequents. The sequence of antecedents or consequents (but not both) may be empty. The intuitive interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the consequents, it means that

$$(A_1 \wedge A_2 \wedge A_3 \wedge \ldots) \quad \supset (B_1 \vee B_2 \vee B_3 \ldots) \tag{2.1}$$

The proof tree starts off with a root node of the form $\vdash A$, where A is the theorem to be proved. PVS proof steps build a proof tree by adding subtrees to leaf nodes as directed by the proof commands. A sequent is *true* if any antecedent is the same as any consequent, if any antecedent is *false*, or if any consequent is *true*. Other sequents can also be recognized as *true*. Once a sequent is recognized as *true*, that branch of the proof tree is terminated. The goal is to build a proof tree whose branches have all been terminated in this way.

The figure 2.2 is an example of the proof tree.

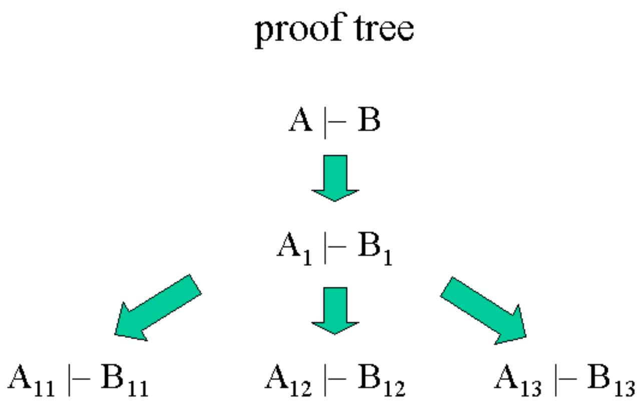


Figure 2.2: ProofTree

In this proof tree the goal is breaking up to subgoals. Validity of subgoals must imply parent goal validity.

A PVS proof focuses on some sequent that is a leaf node in the current proof tree. This is the sequent that is displayed by the PVS prover while waiting for the user's command. The numbers in brackets, e.g.,$[-3]$, and braces, e.g., $\{3\}$, before each formula in the displayed sequent are used to name the corresponding formulas. The formula numbers in square brackets (e.g., $[-3]$ above) indicate formulas that are unchanged in a subgoal from the parent goal where the numbers in braces (e.g., {2} in the example above), are those formulas that are either new or different from those of the parent sequent.

2.2 Sequent Representation Of Proof

Each goal or subgoal in a PVS proof attempt is a sequent of the form $\Gamma \vdash \Delta$, where Γ is a sequence of *antecedent* formulas and Δ is a sequence of *consequent* formulas.

PVS *interactive* commands allow the user to shift the proof (using the postpone command) to a sibling of the current sequent (if any), or to abandon (using the fail or undo command) a portion of the proof containing the current sequent in order to return to some ancestor node representing an earlier point in the proof. For example, one proof step (called **split** in PVS) takes a sequent of the form

$$\Gamma \vdash A \wedge B \tag{2.2}$$

where Γ is any sequence of formulas and creates the pair of child sequents.

$$\Gamma \vdash A \quad and \quad \Gamma \vdash B \tag{2.3}$$

Below is the tree representation of formula (2.2):

$$\begin{tikzcd}
 & \Gamma \arrow[dl, dash] \arrow[dr, dash] & \\
\Gamma \vdash A & & \Gamma \vdash B
\end{tikzcd}$$

A PVS proof command when applied to a sequent provides the means to construct proof trees. These commands can be used to introduce lemmas, expand definitions, apply decision procedures, eliminate quantifiers, and so on; they affect the proof tree, and are saved when the proof is saved. Proof commands may be invoked directly by the user, or as the result of executing a strategy.

The proof commands that define the PVS logic are called the *primitive rules*; they either recognize the current sequent as true and terminate that branch of the proof tree, or they add one or more child nodes to the current sequent and then focus on one of these children. PVS strategies are combinations of proof steps that can add a subtree of any depth to the current node (i.e., the step may invoke sub steps and so on).

PVS proof commands can:

- Cause control to be transferred to next proof sequent in the tree (**postpone**);
- Undo a subtree by causing control to move up to some ancestor node in the proof tree (**undo**);
- Prove the *current sequent* causing control to move to the next remaining leaf sequent in the tree;
- Generate subgoals so that control moves to the first of these subgoals;
- Leave the proof tree unchanged while proving some useful status information.

A proof is completed when there are no remaining unproved leaf sequents in the proof tree. The resulting proof script is saved and can be edited and reused on the same or a different conjecture.

An Example Proof

A simple example of a PVS proof using induction to show that when given two functions f and g on the natural numbers, the sum of the first n values of f and g is the same as the sum of the first n values of the function $\lambda n : f(n) + f(g)$

The theory `sum` below defines the operator `sum` and states the desired theorem as `sum` plus. The main goal of proof is displayed in the *pvs* buffer followed by a Rule? prompt. The user commands are typed in at this prompt.

```
sum: THEORY
  BEGIN
    n: VAR nat
    sum(n): RECURSIVE nat=
    (IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF)
    MEASURE (LAMBDA n:n)
    closed_form:THEOREM sum(n)= (n* (n+1))/2
   END sum
```

PVS starts up by showing :

```
 sum_plus :



    |----------
       {1}  (FORALL (f: [nat -> nat], g: [nat -> nat], n: nat):
              sum((LAMBDA (n: nat): f(n) + g(n)), n) = sum(f, n) + sum(g, n))

 Rule? (skolem!)
```

The first command, `skolem!`, introduces `Skolem` constants `f!1`, `g!1`, and `n!1` for the universally quantified variables in the theorem.

```
 PVS replies as:
 Skolemizing,
 this simplifies to:
 sum_plus :

   |----------

  {1}    sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
             = sum(f!1, n!1) + sum(g!1, n!1)
 Rule?  (lemma "nat_induction")
```

The second command, `lemma`, introduces the induction scheme for natural numbers `nat- induction` as an antecedent formula. This induction scheme is proved as a lemma in the theory natural numbers in the PVS.

```
 Applying nat_induction where
 this simplifies to:
 sum_plus :
  {-1}   (FORALL (p: pred[nat]):
            (p(0) AND (FORALL (j: nat): p(j) IMPLIES p(j + 1)))
                 IMPLIES (FORALL (i: nat): p(i)))
       |-------
   [1]     sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
               = sum(f!1, n!1) + sum(g!1, n!1)
```

The next step is to instantiate the induction scheme with a suitable induction predicate.

```
  Rule? (inst - "(LAMBDA n: sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
             = sum(f!1, n) + sum(g!1, n))")
  Instantiating the top quantifier in - with the terms:
    (LAMBDA n: sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
       = sum(f!1, n) + sum(g!1, n)),
this simplifies to:
```

The `inst` command generates a subgoal where the universally quantified variable p has been replaced by the given induction predicate.

```
  sum_plus :


 {-1}  ((LAMBDA n:
          sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
              = sum(f!1, n) + sum(g!1, n))(0)
             AND
            (FORALL (j: nat):
               (LAMBDA n:
                   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
                        = sum(f!1, n) + sum(g!1, n))(j)
                        IMPLIES
                   (LAMBDA n:
                      sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
                           = sum(f!1, n) + sum(g!1, n))(j+ 1)))


             IMPLIES
            (FORALL (i: nat):
                (LAMBDA n:
                    sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n)
                    = sum(f!1, n) + sum(g!1, n))(i))
  |-------
[1]      sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
             = sum(f!1, n!1) + sum(g!1, n!1)



 Rule?  (beta)
 Applying beta-reduction,
 this simplifies to:

 sum_plus :


{-1} (sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 0) = sum(f!1, 0) +
           sum(g!1, 0)
           AND
          (FORALL (j: nat):
          sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j)
          = sum(f!1, j) + sum(g!1, j)
          IMPLIES sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j + 1)
          = sum(f!1, j + 1) + sum(g!1, j + 1)))

     IMPLIES
 (FORALL (i: nat):
    sum((LAMBDA (n: nat): f!1(n) + g!1(n)), i)
       = sum(f!1, i) + sum(g!1, i))
 |-------
[1]  sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
         = sum(f!1, n!1) + sum(g!1, n!1)
```

Apply the conjunctive splitting command `split` to the goal yields three subgoals. The first goal is to demonstrate that the conclusion of the instantiated induction scheme implies the original conjecture following the introduction of `Skolem` constants. The second subgoal is the base case, and the third subgoal is the induction step. The first subgoal is easily proved by using the heuristic instantiation command `inst?`.

```
  Rule?  (split)
  Splitting conjunctions,
   this yields 3 subgoals:
  sum_plus.1 :
 {-1}  (FORALL (i: nat):
     sum((LAMBDA (n: nat): f!1(n) + g!1(n)), i) = sum(f!1, i) + sum(g!1, i))
    |-------

 [1]     sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
```

```
              = sum(f!1, n!1) + sum(g!1, n!1)

  Rule?   (inst?)
  Found substitution:
  i gets n!1,
  Using template: sum((LAMBDA (n: nat): f!1(n) + g!1(n)), i) =
          sum(f!1, i) + sum(g!1, i)

Instantiating quantified variables,
This completes the proof of sum_plus.1.
```

The second subgoal, contains a formula numbered 2 which was used for the first subgoal proved above. This formula can be suppressed with the hide command. The hidden formulas can be examined using the PVS command M-x show-hidden-formulas.

```
  sum_plus.2 :

  |-------
  {1} sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 0) = sum(f!1, 0) + sum(g!1, 0)
  [2] sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
         = sum(f!1, n!1) + sum(g!1, n!1)

  Rule? (hide 2)
  Hiding formulas: 2,
  this simplifies to:
  sum_plus.2 :

  |-------
 [1] sum((LAMBDA (n: nat): f!1(n) + g!1(n)), 0) = sum(f!1, 0) + sum(g!1, 0)
```

The formula numbered 1 is easily proved by expanding the definition of `sum` using the expand command. This command uses the PVS decision procedures to simplify the definition of `sum` and to reduce the equality to TRUE.

```
 Rule? (expand "sum")
 Expanding the definition of sum,
 this simplifies to:
 sum_plus.2 :


 |-------
```

```
{1} TRUE

which is trivially true.
```

The remaining subgoal is the induction step. It contains the formula numbered 2.

```
sum_plus. :


  |-------

 {1} (FORALL (j: nat):
       sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j) = sum(f!1, j) + sum(g!1, j)
           IMPLIES sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j + 1)
             = sum(f!1, j + 1) + sum(g!1, j + 1))
 [2]    sum((LAMBDA (n: nat): f!1(n) + g!1(n)), n!1)
            = sum(f!1, n!1) + sum(g!1, n!1)


Rule? (hide 2)
Hiding formulas: 2,
this simplifies to:
sum_plus. :


  |-------
[1] (FORALL (j: nat):
     sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j) = sum(f!1, j) + sum(g!1, j)
         IMPLIES sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j + 1)
           = sum(f!1, j + 1) + sum(g!1, j + 1))
```

The `skosimp` command, is a compound of the `skolem!` and `flatten` commands, the resulting simplified sequent contains an antecedent formula, the induction hypothesis, and a consequent formula, the induction conclusion.

```
  Rule?   (skosimp)
  Skolemizing and flattening,
  this simplifies to:
  sum_plus.3 :

  {-1}   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j!1)
            = sum(f!1, j!1) + sum(g!1, j!1)
```

```
    |-------
 {1}  sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j!1 + 1)
          = sum(f!1, j!1 + 1) + sum(g!1, j!1 + 1)
```

By applying the expand command selectively to expand occurrences of sum on the consequent side, a true sequent is obtained.

```
 Rule?  (expand "sum" +) Expanding the
definition of sum, this simplifies to:

  sum_plus.3 :
  [-1]   sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j!1)
             = sum(f!1, j!1) + sum(g!1, j!1)

  |-------
  {1}  sum((LAMBDA (n: nat): f!1(n) + g!1(n)), j!1)
          = sum(f!1, j!1) + sum(g!1, j!1)

 which is trivially true.
 This completes the proof of sum_plus.3.
```

This successfully completes the proof attempt.

3 PVS Languages

3.1 Theories

A PVS specification consists of a collection of *theories*. A theory consists of a sequence of declarations, which provide names for types, constants, variables and formulas. A theory can build on other theories: for example, a theory for ordered binary trees could build on the theory of binary trees. Also a theory can be parametric in certain specified types and values, e.g., a theory of queues can be parametric in the maximum queue length, and a theory of ordered binary trees can be parametric in the element type as well as ordering relation. Figure 2.3 is the general template form of the PVS theories. Theorems must be proved in the context of former declarations and definitions. Specifications in PVS are built from *theories*, which provide reusability and structuring. A theory consists of a *theory identifier*, a list of formal *parameters*, an EXPORTING clause, an *assuming part*, a *theory body* and an ending id. The syntax for theories in shown in Appendix A (PVS grammar).
Everything is optional expect the identifiers and the keywords. The simplest theory has the form:

```
name1 :THEORY
  BEGIN
  END name1
```

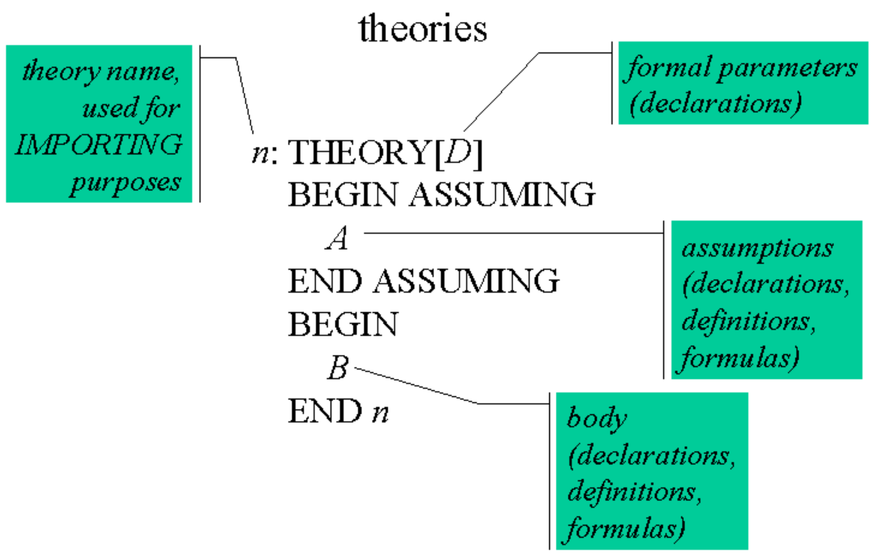

theories
theory name, used for IMPORTING purposes
formal parameters (declarations)
n: THEORY[D]
BEGIN ASSUMING
A
END ASSUMING
BEGIN
B
END n
assumptions (declarations, definitions, formulas)
body (declarations, definitions, formulas)

Figure 2.3: PVS Theories Template

The formal parameters, assuming and theory body consist of declarations and `IMPORTING`. Names declared in a theory may be available to other theories in the same context by means of the `EXPORTING` clause. Names exported by a given theory may be imported into a second theory by means of `IMPORTING` clause. The assuming part precedes the theory part, so the theory part may refer to entities declared in the assuming part. The purpose of assuming part is to provide constraints on the use of the theory, by means of the *ASSUMING*.

Theory Identifiers

The theory identifier introduces a name for a theory. In the PVS system, the sets of theories from a *context*. Within the context theory names must be unique. Prelude is the initial available context, which provides the Boolean operators, equality, and the `real`, `rational`, `integer`, and `naturalnumbers` types and their associated properties. The only differences between the prelude and user defined theories is that the prelude is automatically imported in the theory, without requiring an explicit `IMPORTING` clause. The identifier at the end, must match the theory identifier, otherwise there will be an error message in the PVS system.

Theory Parameters

Theory parameters can be types, subtypes or constants, and `IMPORTING`s. The parameters must have unique identifiers. The parameters are ordered, allowing later parameters to refer to earlier ones.

3.2 PVS Type System

PVS is a strongly typed specification language. The simply typed fragment includes types constructed from the base types by the function and product type constructions, and expressions constructed from the constants and variables by means of application, abstraction, and tupling. Expressions are checked to be well typed under a *context*, which is a partial function that assigns a kind (one of the `TYPE, CONSTANT` or `VARIABLE`), and a type to the constant and variable symbols. There are type constructors for *subtypes*, *function types*, *tuple types*, and *record types*. Function, record, and tuple types may also be dependent.

Simple Types

PVS includes primitive types such as booleans or real, and classic constructors for forming functions and tuples types. For example:

- `[real, real -> bool]` is the types of functions from pairs of reals to the booleans.
- `[nat, nat, nat]` is the type of triples of natural numbers.

There are also other constructions for record types and built in support for abstract data types.

Subtypes

Subtyping is one of the main features of the PVS specification language. Subtyping in PVS corresponds to the set theoretic notation of subset. In the simple type fragment, each type corresponds to a set of values that is structurally different from the set of values for another type, so that a term has at most one type. Subtyping makes it possible to introduce the natural numbers as a subtype of the reals. With subtyping a term can have several possible types, but the typechecking function τ may return only a single type. The function τ can consider to return a natural canonical type of an expression that is given by the declaration of the symbols in the expression. If the expression is used in a context where the expected type is a supertype of its canonical type, then the type correctness is straightforward. If the expected type is a subtype that is compatible with the canonical type of the expression, then typechecking generates a proof obligation asserting that the expression satisfies the predicate constraints imposed by expected type.
Given an arbitrary function p of type `[t->bool]`, one can define the subtype of t consisting of all the elements which satisfy p. This type is denoted by $\{x : t|p(x)\}$ or equivalently (p). More generally, subtypes can be constructed using arbitrary boolean expressions. For example:

```
nzreal:TYPE = {x:real |x/=0}
```

declares the type `nzreal` whose elements are the non-null reals. Subtypes can also be declared as follows:

```
s: TYPE FROM t;
```

This defines s as an uninterpreted subtype of t. With this declaration PVS automatically associates a predicate $s_(pred)$: $[t- > bool]$ characteristic of s: the two expressions s and $s_(pred)$ denote the same subtype of t. By default, PVS does not assume that types are non empty but the user can assert that types are inhabited as follows:

```
s: NONEMPTY_ TYPE FROM t
```

Function Types

Function types are introduced by specifying their domain and range types: for example binary arithmetic operations such as addition and multiplication have type `[[real, real] -> real]`, and can also written as `real, real -> real]`.

Furthermore function types have three equivalent forms:

- $[t_1$,..., t_n-> $t]$
- FUNCTION $[t_1$, ..., t_n -> $t]$
- ARRAY $[t_1$, *ldots*,-> $t]$

where each t_i is a type expression. An element of this type is simply a function whose domain is the sequence of types t_1, ..., t_n, and whose range is t. A function type is empty if the range is empty and the domain is not. There is no difference in meaning between these three forms; they are provided to support different intensional uses of the type. For example the two forms `pred [t]` and *setof [t]* are both provided in the prelude as shorthand for $[t- > bool]$. There is no differences in semantics, as sets in PVS are represented as predicates. The different keywords are provided to support different intentions; `pred` focuses on properties while `setof` tends to emphasize elements. A function type $[t_1, \ldots, t_n- > t]$ is a subtype of $[s1, \ldots, s_m- > s]$ iff s is a subtype of t, $n = m$, and $1 \leq i \leq n$.

Record Types

Record types are a list of label type pairs, for example, `[# age:nat, years- employed:nat #]`. The a_i are called record accessors or fields and the t_i are types. Record types are similar to tuple types, except that the order is unimportant and accessors are used instead of projections. Record types are empty if any of the component types is empty.

Tuple Types

Tuples, such as `[nat, bool]`, are similar to records excepts that fields are accesses by the order of their appearance rather than by labels. Tuple types have the form $[t_1, \ldots, t_n]$, where the t_i are type expressions. Note that the 0-array tuple is not allowed. Elements of this type are tuples whose components are elements of the corresponding types. For example, (1, TRUE, (LAMBDA (x:int): x+1)) is an expression of type $[int, bool, [int- > int]]$. A tuple type is empty if any of its component types is empty. Function type domain and tuple types are closely related, as the types $[t_1, \ldots, t_n- > t]$ and $[[t_1, \ldots, t_n]- > t]$ are equivalent.

Dependent Types

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

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

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

3.3 Declarations

Entities of PVS are introduced by means of *declarations*, which are the main constituents of PVS specifications. Declarations are used to introduce type, variables, constants and formulas. Each declaration has an *identifier* and belongs to a unique theory. Declarations also have a body which indicates the kind of the declaration and provides the signature and definition of the declaration. Declarations introduced in one theory may be referenced in another by means of the `IMPORTING` and `EXPORTING` clauses. The `EXPORTING` clauses of a theory indicates those entities that may be referenced from outside the theory. The `IMPORTING` clauses provide access to the entities exported by another theory. Declaration consists of an *identifier*, an optional list of *bindings*, and a *body*. The body determines the kind of the declaration, and the binding and body together determine the signature and definition of the declared entity. Multiple declarations may be given in compressed form in which one body is specified for multiple identifiers; for example:

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

The above is equivalent to :

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

3.4 Type Declarations

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

- *Uninterpreted type declarations*: T: TYPE
- *Uninterpreted subtype declaration* S: TYPE FROM T
- *Interpreted type declarations*: T: TYPE= [int->int]
- *Enumeration type declarations*: T: TYPE= {r, g, b}

Uninterpreted type declarations

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

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

This is introduces three new pairwise disjoint types.

Uninterpreted subtype declaration

Uninterpreted subtype declaration are the form of:

```
s: TYPE FROM t
```

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

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

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

Interpreted type declarations

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

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

introduces the type name `intfun` as an abbreviation for the type of functions with integer domain and range. Interpreted type declarations may be given parameters. For example, the type of integer subranges may be given as:

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

and `subrange` with two integer parameters may subsequently be used wherever a type is expected. Note that `subrange` may be used to declare a different type without any ambiguity, as long as the number of parameters is different.

Enumeration type declarations

Enumeration type declarations are of the form:

```
enum: TYPE= {e_1,..., e_n}
```

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

$$\begin{array}{c}\text{enum: DATATYPE}\\ e_1 : e_1?\\ \vdots \\ e_n : e_n?\\ \text{END enum}\end{array}$$

Types in specification languages are often interpreted as sets of values, and it has a association of subtype with subset: one type is a subtype of another if the set interpreting the first is a subset of that interpreting the second. *Predicate* subtypes provide a characterization by associating a predicate or property with the subtype. For example, the natural numbers are subtype of the integers characterized by the predicate"greater than or equal to zero". Predicate subtypes can help make specifications more succinct by allowing information to be moved into the types, rather than stated repeatedly in conditional formulas. For example, instead of:

$$\forall(i,j:int): i \geq 0 \wedge j \geq 0 \supset i+j \geq i$$

where $\supset$ is logical implication, we can write

$$\forall(i,j:nat): i+j \quad \geq i$$

because $i \geq 0$ and $j \geq 0$ are recorded in the type `nat` for i and j. Theorem proving can be required in typechecking some constructions involving predicate subtypes. For example, if `half` is a function that requires an `even` number as its argument, then the formula:

$$\forall(i:int): half(i+i+2) = i+1$$

is well typed only if we can prove that the integer expression $i+i+2$ satisfies the predicate for the subtype `even`: that is, if we can discharge the following proof obligation.

$$\forall(i:int) \quad \exists \quad (j:int): i+i+2 = 2 \times j$$

In the theory definition, types can be defined starting from base types (booleans, numbers, etc) using the function, record and tuple type constructions. The language supports modularity and reuse by the means of parameterized *theories*, and has a type system including the notation of a *predicate subtype.*

PVS specifications are structured into parameterized theories that can have constraints attached to the parameters. Constraints can also be attached to the types in a PVS specification. For instance, the division operation is typed so that it is constrained to nonzero divisors. Constraints are attached to types in PVS using *predicate subtypes*, so that the signature for division can be given as:

```
    nonzero : TYPE= {x: rational | x/=0}
      /: [rational, nonzero -> rational]
```

where `rational` is the type of rational numbers, and `nonzero` is defined here to be the subtype of the rational numbers different from zero. When the PVS type checker is invoked on the formula:

```
    x/=y IMPLIES (y-x)/(x-y)<0
```

it recognizes the divisor `(x-y)` must be shown to be nonzero and generates a TCCs of the form:

```
   (FORALL (y, x:rational): x /= y IMPLIES (x-y) /=0)
```

3.5 Partial Functions

PVS only works with total functions: given sets A and B and a relation $R \subseteq A \times B$, then R is a total function from A to B, if for every member of A there is exactly one member of B that is related to it. In order to deal with the partial functions in PVS, there should be a restriction in the domain.
For example: $\{a, b : real | b > a\}$

Since functions, are total in PVS, the relations can be used to represent the partial functions. There are simple functions that turn a PVS function $f\colon [A- > B]$ into a relation $R\colon [A, B- > bool]$ which is a total function, and conversely, a total relation $R\colon [A, B- > bool]$ into a PVS function $f\colon [A- > B]$.

Overview

This chapter gave a general introduction to PVS and the initial steps to start a proof in PVS: creating the specification, parsing, type checking and proving parts. Furthermore the structure of the PVS proof and general concepts about proof goals and an example proof were discussed. The last part of the chapter covered the PVS languages.

Chapter 3

Parser

This chapter deals with the concepts of the parser and the lexical structure in PVS and the example of lexical structure according to the PVS grammar in JavaCC.

A parser for a grammar is a program which takes the language string as its input and produces either a corresponding parse tree or an error. The rules which tell whether a string is a valid program or not are called the syntax and the rules which give meaning to programs are called the semantics of a language. When a string representing a program is broken into a sequence of substrings, such that each substring represents a constant, identifier, operator, keyword, etc of the language, these substrings are called the *tokens* of the language. Parsing a PVS specification has two main parts:

- Check that the specification is syntactically correct (e.g., satisfy the PVS grammar).
- Build the internal data structures representing a theory.

1 Lexical Analysis

To translate a program from one language to another, a compiler must first understand its structure and meaning. The analysis is usually broken up into:

- Lexical analysis: breaking the input program into individual words or tokens.
- Syntax analysis: parsing the phrase structure of the program.
- Semantic analysis: defining the program's meaning.

The function of a lexical analyzer is to read the input stream representing the source program, one character at time, and to translate it into valid tokens. The tokens in a language are represented by a set of *regular expressions*; it is thus possible to implement lexical analyzer by *deterministic finite automata*. A regular expression specifies a set of strings to be matched. It contains text characters and operator characters. The tokens which are represented by a regular expression are recognized in an input string by means of a finite automaton. The lexical analysis can store all the recognized tokens in an intermediate file and give it to the parser as an input. The PVS specification are text files, each composed of a sequence of lexical elements which in turn are made up of characters. The programming languages which will use to develop of the PVS parser is JavaCC (stands for Java Compiler Complier). Figures 3.1 and 3.2 show the relationship between a JavaCC generated lexical analyzer (token manager) and a JavaCC generated parser. The token manager reads in a sequence of characters and produces a sequence of tokens. The parser uses the sequence of tokens, analyses its structure, and produces the parse tree.

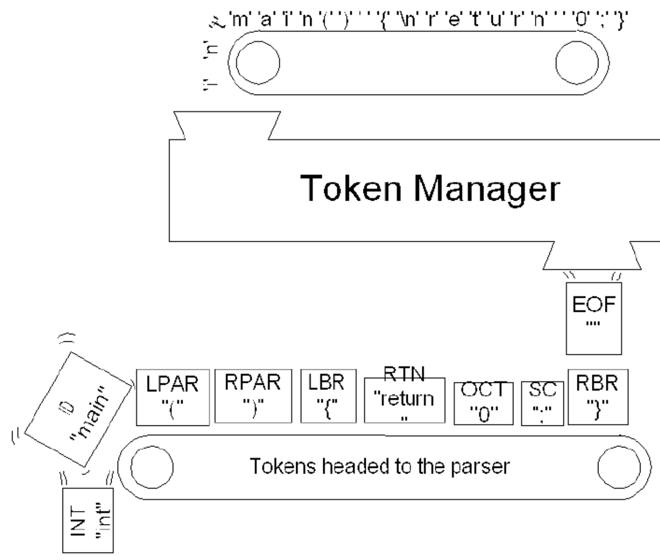


Figure 3.1: Token Manager

1.1 The Lexical Structure In PVS

The lexical elements of PVS are the identifiers, reserved words, special symbols, numbers and comments. Identifiers are composed of letters, digits, and the characters _ and ?; they begin with a letter. Identifiers in PVS are different for upper and lower case; e.g., `FOO`, `Foo` and `foo` are different identifiers.
The lexical syntax of PVS is as follows:

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

The iteration of a clause one or more times is indicated by a plus sign. Repetition zero or more times is indicated by an asterisk instead of the plus sign. Reserved words are not case sensitive. Identifiers may have reserved words embedded in them. For example, `ARRAYALL` is a valid identifier and will not be confused with two embedded reserved words.

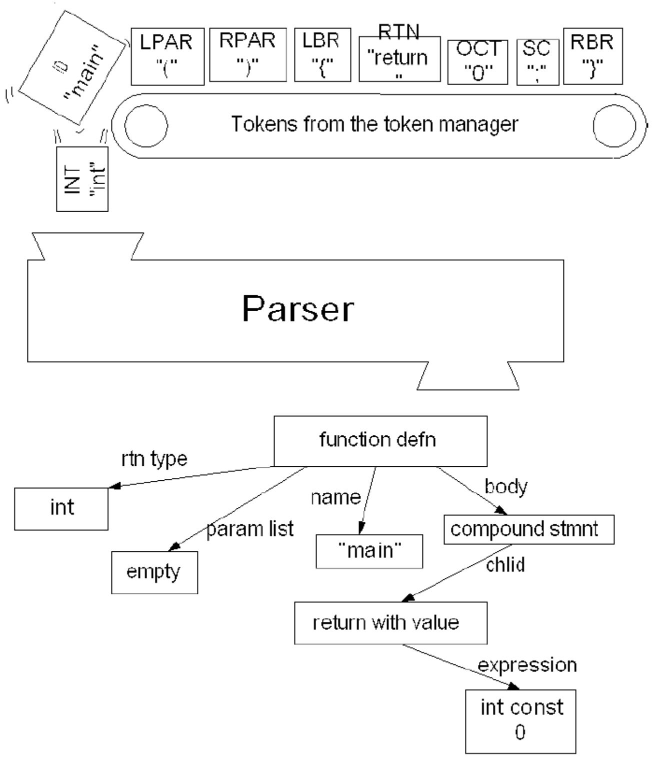


Figure 3.2: Parser

1.2 An Example For PVS Lexical Structure

Appendix A provides the whole PVS grammar, which is used to generate the parser and lexical analyzer. Following is one example for generating the lexical analyzer according to the PVS grammar for specification part and the corresponding JavaCC code. The PVS grammar for specification part:

```
Specification


specification   ::=  {theory |datatype} '+'
theory          ::= id [theory-formals] :THEORY
                 [exportings]
                 BEGIN
                 [assuming-part]
                 [theory-part]
                 END id
theory-formals  ::= [theory-formal++ ',']
theory-formal   ::=[(importing)] theory-formal-decl
theory-formal-decl ::= idops : {TYPE | NONEMPTY_TYPE |TYPE +}
                              [FROM type-expr]
                    | idops :type-expr
```

The JavaCC code corresponding with the grammar should be as follows:

Specification part, which is any number of either a Theories or a Datatypes.

```
TreeRoot Specification() :
{
        Node                    v;
        java.util.Vector        vect    = new java.util.Vector();
}
{
        ( ( LOOKAHEAD(3) v=Theory() | v=Datatype() ) { vect.addElement(v); } )
        + <EOF>
        {
                return new TreeRoot(vect);
        }
}
```

If it is Theory part then:

```
NodeTheory Theory() :
{
        Node                    a;
        Node                    b = new NodeEmpty();
        Node                    c = new NodeEmpty();
        Node                    d = new NodeEmpty();
        Node                    e = new NodeEmpty();
        Node                    f;
}
{
        a=Id() [ b=TheoryFormals() ] <COLON> <THEORY>
        [ c=Exporting() ]
        <BEGIN>
                [ d=AssumingPart() ]
                [ e=TheoryPart() ]
        <END> f=Id()
        {
                return  new NodeTheory(a,b,c,d,e,f);
        }
}
```

The theory formals part will be:

```
NodeTheoryFormals TheoryFormals() :
{
        Node                    v;
        java.util.Vector        vect = new java.util.Vector();
}
{
        <LBRAC> v=TheoryFormal() {vect.addElement(v);}
       (<COMMA> v=TheoryFormal(){vect.addElement(v);})* <RBRAC>
        {
                return new NodeTheoryFormals(vect);
        }
}
```

The part for the theory formal will be:

```
NodeTheoryFormal TheoryFormal() :
{
        Node                    a = new NodeEmpty();
        Node                    b;
}
{
        [ LOOKAHEAD(2) <LPARAN> a=Importing() <RPARAN> ] b=TheoryFormalDecl()
        {
                return new NodeTheoryFormal(a,b);
        }
}
```

The theory formal declaration part:

```
NodeTheoryFormalDecl TheoryFormalDecl() :
{
        Node    a;
}
{
        LOOKAHEAD(3)   a=TheoryFormalType()   {return new NodeTheoryFormalDecl(a);}
|                      a=TheoryFormalConst()  {return new NodeTheoryFormalDecl(a);}
}

NodeTheoryFormalType TheoryFormalType() :
{
        Node                    a;
        Token                   t;
        Node                    b = new NodeEmpty();
}
{
        a=Ids() <COLON> (t=<TYPE>|t=<NONEMPTY_TYPE>|t=<TYPEP>)
         [ <FROM> b=TypeExpr() ]
        {
                return new NodeTheoryFormalType(a,new NodeToken(t),b);
        }
}
```

Theory formal constant part:

```
NodeTheoryFormalConst TheoryFormalConst() :
{
        Node                    a;
        Node                    b;
}
{
        a=IdOps() <COLON> b=TypeExpr()
        {
                return new NodeTheoryFormalConst(a,b);
        }
}
```

It was the whole JavaCC code for the `specification` part. The details of the look ahead and PVS parser will be given in chapter6 (Design).

Overview

This chapter covered the main concepts of the parser and lexical analysis. It also gave the description of PVS lexical structure and JavaCC. The last part of the chapter gave an example for the specification part in PVS grammar and the corresponding JavaCC code.

Chapter 4

Theorem Prover

This chapter focuses on the *theorem prover* and the basic concepts behind the *logic system* and theorem prover in PVS. The last part of the chapter will give a short introduction to derived rules as a tactic for theorem proving in PVS.
A theorem prover consists of a proof language and a mechanical proof checking mechanism. There are different ways of defining a proof system. In general a proof is a finite sequence of a formulae, where the last formula is the theorem to be proved, and each formula in the sequence is either an axiom or derived from previous formulas by rules of logical inference.
The life cycle to a mechanically proof checking is:

- Exploratory phase of proof development: mainly interest is in debugging the specification and putative theorems and in testing and revising the key, high level ideas in the proof. An important requirement in this phase is early and useful feedback when a purported theorem is false.
- Development phase: construct the proof in large steps. Once the basic intuitions have been acquired and the formalization is stable, the proof checking enters a development phase.
- Generalization: where the proof should carefully analyze and finish, weaken and generalize the assumptions, extract the key insights and proof techniques, and make it easier to carry out subsequent verification of a similar nature.
- Maintenance: is a special application of generalization, where we adapt a verification to slightly changed assumptions or requirements.

The goal of PVS proof checker is to support the efficient development of readable proofs in all stage of the proof development life cycle. The proof checker implements a small set of powerful primitive inference rules, a mechanism for composing these into proof strategies, a facility for rerunning proofs, and another to check that all secondary proof obligations (such as type correctness conditions) have been discharged.

1 Logic

A *logic system* consists of a language of *symbols*, a proof system for manipulating the symbols and an associated model theory giving meaning to the symbols. Symbolic logic considers languages whose essential purpose is to symbolize reasoning encountered in mathematics. There are different sorts of logic: Higher order, first order and propositional Logic.

2 Propositional Logic

In the propositional logic all statements are definitely true or false. They don't depend on any variables or parameters. Propositional logic deals with truth values and logical connectives 'and', 'or', 'not', etc.

2.1 Syntax Of Propositional Logic

The formulas of propositional logic are strings of symbols build up from basic elements called ***propositions*** which are denoted P, Q, R,..., including the truth constants T and F. A formula consisting of a propositional symbol is called *atomic*. Formula are constructed from atomic formula using the logical connectives $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\leftrightarrow$ (if and only if). These are listed in order of precedence; $\neg$ is highest. The syntax of propositional logic is:

$$\text{proposition} ::= p_0|p_1|p_2|\ldots$$

$$\text{bin-connective} ::= \wedge|\vee|\rightarrow|\leftrightarrow$$

$$\text{formula} ::= \text{proposition} \mid \neg\ \text{formula} \mid \text{formula bin-connective formula}$$

2.2 Semantics Of Propositional Logic

Propositional logic is a formal language. Each formula has a meaning (or semantic) either true or false relative to the meaning of the propositional symbols.

An *interpretation*, or *truth assignment*, for a set of formula is a function from its set of propositional symbols to $\{t, f\}$. An interpretation *satisfies* a formula if the formula evaluates to t under the interpretation.

A set S of formula is *valid* if every interpretation for S satisfies every formula in S. A set of formula is *satisfiable* if there is some interpretation for S that satisfies every formula in S.

A set of formula is *unsatisfiable* if it is not satisfiable. A set S of formula is *entails* A if every interpretation that satisfies all elements of S, also satisfies A. It write $S \vdash A$.

Formula A and B are *equivalent*, $A \simeq B$, provide $A \vdash B$ and $B \vdash A$. In propositional logic, a valid formula is also called a *tautology*. Some examples of valid formulas:

- The formulas $A \rightarrow A$ and $\neg(A \vee \neg A)$ are valid for every formula A.
- The formulas P and $P \vee (P \rightarrow Q)$ are satisfiable: they are both true under the interpretation that maps P and Q to `t`. But they are not valid: they are both false under the interpretation that maps P and Q to `f`.
- The formula $\neg A$ is unsatisfiable for every valid formula A.

2.3 Equivalences

Below is the list of the basic equivalences of the propositional logic. There can be useful for transforming one proposition into an equivalent one.

Rule name	Rule1	Rule2
Idempotency	$A \wedge A \simeq A$	$A \vee A \simeq A$
Commutative rules	$A \wedge B \simeq B \wedge A$	$A \vee B \simeq B \vee A$
Associative rule	$(A \wedge B) \wedge C \simeq A \wedge (B \wedge C)$	$(A \vee B) \vee C \simeq A \vee (B \vee C)$
Distributive rules	$A \vee (B \wedge C) \simeq (A \vee B) \wedge (A \vee C)$	$A \wedge (B \vee C) \simeq (A \wedge B) \vee (A \wedge C)$
De Morgan rules	$\neg(A \wedge B) \simeq \neg A \vee \neg B$	$\neg(A \vee B) \simeq \neg A \wedge \neg B$
Connectives rules	$A \leftrightarrow B \simeq (A \rightarrow B) \wedge (B \rightarrow A)$	$A \rightarrow B \simeq \neg A \vee B$
Negation rules	$\neg(A \rightarrow B) \simeq A \wedge \neg B$	$\neg(A \leftrightarrow B) \simeq (\neg A) \leftrightarrow B \simeq A \leftrightarrow (\neg B)$

In propositional, logic for every equivalence $A \simeq B$ there is another equivalence $\acute{A} \simeq \acute{B}$, where $\acute{A}$ and $\acute{B}$ are derived from A and B by exchanging $\wedge$ with $\vee$ and `t` with `f`.

2.4 Normal Forms

The language of propositional logic is redundant: many of the connectives can be defined in term of others. By repeatedly applying certain equivalences, we can transform a formula into a *normal form*. A typical normal form eliminates certain connectives entirely, and uses others in a restricted manner. The restricted structure makes the formula easy to process, although the normal form may be exponentially larger than the original formula. A *literal* is a proposition or its negation. Let K, L, $\acute{L}$,... stand for literals:

- A formula is in *Negation Normal Form* (NNF) if the only connectives in it are $\wedge$, $\vee$ and $\neg$, where $\neg$ is only applied to a proposition.
- A formula is in *Conjunctive Normal Form* (CNF) if it has the form $A_1 \wedge \ldots \wedge A_m$, where each A_i is a maxterm (A maxterm is a literal or a disjunction of literals).
- A formula is in *Disjunctive Normal Form* (DNF) if it has form $A_1 \vee \ldots \vee A_m$, where each A_i is a minterm (A minterm is a literal or conjunction of literals).

2.5 Examples

P is an atomic formula so it will be in all normal forms: NNF, CNF and DNF. The example for CNF will be like:

$$(P \vee Q) \wedge (\neg P \vee Q) \wedge R$$

and by exchanging $\wedge$ and $\vee$ the formula will be in the DNF form:

$$(P \wedge Q) \vee (\neg P \wedge Q) \vee R$$

There are some rules for translation to normal forms, every formula can be translated into an equivalent formula in NNF, CNF or DNF by following steps:

`Step 1.` Eliminate $\leftrightarrow$ and $\rightarrow$ repeatedly applying the following equivalences:

$$A \leftrightarrow B \simeq (A \rightarrow B) \wedge (B \rightarrow A)$$
$$A \rightarrow B \simeq \neg A \vee B$$

`Step 2.` More negation inwards until it applies only to atoms, repeatedly replacing by the equivalences: Now the formula is in Negation Normal Form:

$$\neg\neg A \simeq A$$
$$\neg(A \wedge B) \simeq \neg A \vee \neg B$$
$$\neg(A \vee B) \simeq \neg A \wedge \neg B$$

`Step 3.` Now the formula is in Negation Normal Form, for getting CNF form, the disjunction must pushed in until it applies to literals, repeatedly use by the equivalences:

$$A \vee (B \wedge C) \simeq (A \vee B) \wedge (A \vee C)$$
$$(B \wedge C) \vee A \simeq (B \vee A) \wedge (C \vee A)$$

These two equivalences are the same, because of the commutatively of the disjunction. Then the following formula is obtained:

$$(A \wedge B) \vee (C \wedge D) \simeq (A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$$

`Step 4.` Simplify the result of the CNF by deleting any maxterm that contains both a literal and a negation of the same literal ($\neg P$ and P), since it is always `t`. Also delete any maxterm that includes another maxterm: $A \wedge (A \vee B) \simeq A$. Finally two maxterms of the form $P \vee A$ and $\neg P \vee A$ can be replaced by A.

$$(P \vee A) \wedge (\neg P \vee A) \simeq A$$

2.6 A Method For Theorem Proving In Propositional Logic

To prove A, reduce it to CNF. If the simplified CNF formula is `t` then A is valid, because each transformation preserves logical equivalences. If the CNF formula is not `t`, then A is not valid. If $A_1 \wedge \ldots \wedge A_m$ is CNF formula and if A is valid then each A_i must also be valid. The literal A_i can be as $L_1 \vee \ldots \vee L_n$, where the L_j are literals, then interpretation I can be made that falsifies every L_j and therefore falsifies A_i. I is defines as follows, for every propositional letter P:

$I(P)$= `f` if L_j is P for some j and $I(P)$= `t` if L_j is $\neg P$ for some j. This definition is correct because there are no literals L_j and L_k such that L_j is $\neg L_k$. If they exist, then simplification would have deleted the disjunction A_i. The method for theorem proving in propositional logic starts by using an if-then-else data structure, an ordering on the propositional letters.

Example

Start with :

$$P \vee Q \rightarrow Q \vee R$$

Step 1, eliminate $\rightarrow$, gives:

$$\neg(P \vee Q) \vee (Q \vee R)$$

Step 2, push negation in, gives:

$$(\neg P \wedge \neg Q) \vee (Q \vee R)$$

Step 3, push disjunctions in, gives:

$$(\neg P \vee Q \vee R) \wedge (\neg Q \vee Q \vee R)$$

Simplifying :

$$(\neg P \vee Q \vee R) \wedge t$$
$$\neg P \vee Q \vee R$$

The interpretation $P \mapsto \texttt{t}$, $Q \mapsto \texttt{f}$, $R \mapsto \texttt{f}$ falsifies the formula, which is equivalent to the original formula. So the original formula is not valid.

3 Binary Decision Trees

The propositional formula can represented by a binary decision tree. A binary decision tree is a rooted, directed tree with two types of vertices, *terminal vertices* and *nonterminal vertices.*

- Each nonterminal vertex v is labeled by a variable *var(v)* and has two successors:
  - *low(v)*: corresponding to the case where the variable v is assigned 0.
  - *high(v)*: corresponding to the case where the variable v is assigned 1.
- Each terminal vertex v is labeled by *value (v)* which is either 0 or 1 (false or true).

Example

A binary decision tree for the below formula:

$$(a \wedge b) \vee (c \wedge d)$$

is shown in the figure 4.1. A dashed line corresponds to value 0, a plain line to value 1. The decision tree is ordered by $a < b < c < d$.

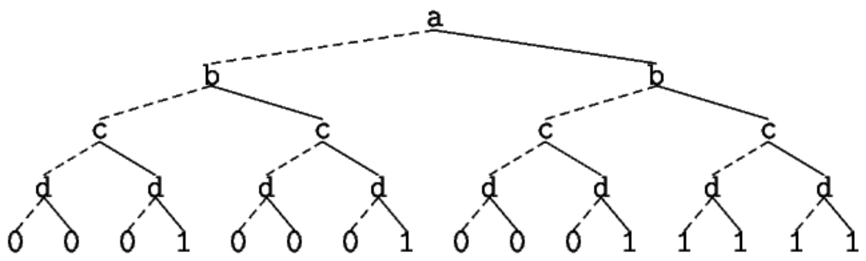


Figure 4.1: BDT Diagram

3.1 Binary Decision Diagram

A BDD is a rooted, directed acyclic graph with two types of vertices, terminal vertices and nonterminal vertices:

- Each nonterminal vertex:
  - is labeled by a variable *var(v)*.
  - it has two successors, *low (v)* and *hight(v)*.
- Each terminal vertex:
  - is labeled by either 0 or 1.

Every BDD with root v determines a boolean function $f_v(X_1, \ldots, X_n)$ in the following manner:

1. If v is a terminal vertex:
   (a) If *(v)*=1 then $f_v(X_1, \ldots, X_n)$=1
   (b) If *(v)*=o then $f_v(X_1, \ldots, X_n)$=0.
2. If v is a nonterminal vertex with *var (v)*=X_i then f_v is the function:

$$f_v(x_1, \ldots, x_n) = (\neg x_i \wedge f_{low(v)}(x_1, \ldots, x_n) \vee (x_i \wedge f_{high(v)}(x_1, \ldots, x_n)))$$

Following are transformation rules to avoid redundancy of the vertices in the diagram:

3.2 The Transformation Rules

1. All duplicate terminals should removed.
2. All duplicate nonterminals should removed:
   (a) If two nonterminals u and v have $var(u) = var(v)$, $low(u) = low(v)$ and $high(u) = high(v)$, then u or v should be eliminated and all the incoming arcs to the other vertex should be redirected.
3. All the redundant tests should be eliminated:
   (a) If nonterminal v has $low(v) = high(v)$, then eliminate v and redirect all incoming arcs to $low(v)$.

3.3 Ordered Binary Decision Diagrams

Binary decision diagram is a directed graph. In the *ordered* binary decision diagram (OBDD) the variables are ordered, they must be tested in order, for example:

$$a_1 < b_1 < \ldots < a_n < b_n$$

There is a corresponding between each propositional formula and a unique OBDD, for a given ordering. The conditions which should satisfy by OBDD are:

- *Ordering*: if P is tested before Q, then $P < Q$.
- *Uniqueness*: identical subgraphs are stored only once.
- *Redundancy*: no test leads to identical subgraphs in the `t` and `f` cases.

Ordering of OBDDs

The size of an OBDD can depend on the variable ordering. If the ordering of $a_1 < b_1 < a_2 < b_2$ is chosen then the OBDD will as the figure 4.2.

In general, finding an optimal ordering for the variables is infeasible: in fact, it can be shown that even checking that a particular ordering is optimal is NP-complete. Furthermore, there are boolean functions that have exponential size OBDDs for any variable ordering.

Several heuristic algorithms have been developed for finding a good variable ordering when such an ordering exists.

- If the boolean function is given by a combinational circuit, then heuristics based on a depth-first traversal of the circuit diagram generally give good results.
- The intuition for these heuristics comes form the observation that OBDDs tend to be small when related variables are close together in the ordering.
- A technique called *dynamic reordering* appears to be useful in those situations where no obvious ordering heuristics apply.
- The reordering method is designed to save time rather than to find an optimal ordering.

In order to know that if two formula are logically equivalent or not, they can be convert to their OBDD form and then by using the uniqueness rule they can be checked.

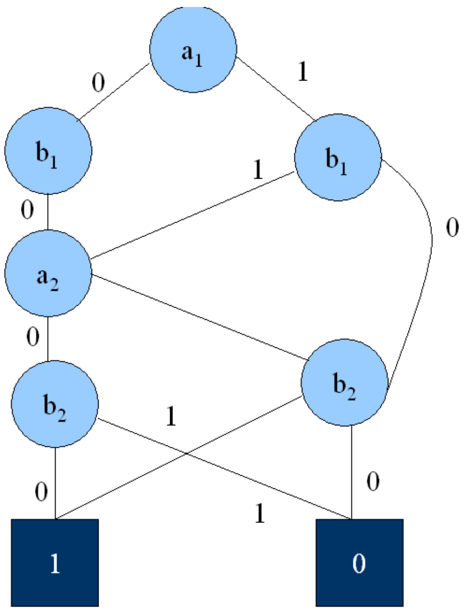


Figure 4.2: OBDD Diagram

4 First Order Logic

First- order logic extends propositional logic to allow reasoning about members (such as numbers) of some non-empty universes. It uses the quantifiers ∀ (for all) and ∃ (there exists). *First-order logic* replaces the set of propositional atoms of propositional logic with sets of *predicates*, *functions*, and *variables*, and allows variable *quantification*.

4.1 Syntax Of First-Order Logic

Terms stand for individual values while formula stand for truth values. A *first-order language* L contains, for all $n \geq o$, a set of n-place *function symbols*, f, g,… and n-place *predicate symbols* P, Q, … These sets may be empty, finite, or infinite.

The terms t, u, … of a first-order language are define recursively as follows:

- A variable is a term.
- A constant symbol is a term.
- If t_1, …, t_n are terms and f is an n-place function symbol then f$(t_1, \ldots, t_2)$.

4.2 Formal Semantic Of First-Order Logic

An interpretation of a language maps its function symbols to actual functions, and its relation symbols to actual relations. For example the predicate 'student' can be map to the set of all students currently enrolled at the university.

Definition

Let l be a first-order language. An *interpretation* τ of l is a pair (D, I). D is a domain of nonempty set. The operation I maps symbols to individuals, functions or sets:

- if c is a constant symbol of l then $I[c] \in D$.
- if f is an n-place function symbol then $I[f] \in D^n \to D$ (which means $I[f]$ is an n-place function on D.
- if P is an n-place relation symbol then $I[P] \subseteq D^n$ (which means $I[p]$ is an n-place relation on D).

$\forall x(p)$, evaluates to *true* if any assignment of a value to x, makes p true. And $\exists x(p)$ evaluates to *true* if some assignment of a value to x, makes p true.

5 The Sequent Calculus

The *sequent calculus*, makes the set of assumptions explicit, thus it is more concrete.

A *sequent* has the form $\Gamma \vdash \Delta$, where Γ and Δ are finite set of formulas. The sequent :

$$A_1 \cdots, A_m \vdash B_1 \cdots, B_n$$

is *true* if $A_1 \wedge \cdots \wedge A_m$ implies $B_1 \vee \cdots \vee B_n$. In other words, we assume that each of $A_1, \cdots, A_m$ are true and try to show that at least one of $B_1, \cdots, B_n$ is true. A *basic* sequent is one in which the same formula appears on both sides, as in $P, B \vdash B, R$. This sequent is true because, if all the formula on the left side are true, then in particular B is; so at least one right-side formula (B again) is true. Every basic sequent might be written in the form:

$$\{A\} \cup \Gamma \vdash \{A\} \cup \Delta$$

where A is the common formula and Γ and Δ are the left-and right-side formula.

The sequent calculus identifies the one-element set $\{A\}$ with its element A and denotes union by a comma. Thus, the usual notation for the general form of a basic sequent is $A, \Gamma \vdash A, \Delta$. Sequent rules are almost always used backwards. We start with the sequent that we would like to prove and, working backwards and reduce it to simpler sequents. Sequents rules are classified as *right* or *left*, indicating which side of the $\vdash$ symbol they operate on.

$$\frac{A}{A \vee B} \ (vi1) \quad \frac{B}{A \vee B} \ (vi2)$$

The analogue of the pair $(vi1)$ and $(vi2)$ is the single rule:

$$\frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B} \quad (\vee r)$$

This breaks down some disjunction on the right side, replacing it by both disjunctions. The sequent rules for propositional logic are:

$$\text{basic sequent}: \ A, \Gamma \vdash A, \Delta$$

Negation rules:

$$\frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta} \quad (\neg l) \quad \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B} \quad (\wedge r)$$

Conjunction rules:

$$\frac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \quad (\wedge l) \quad \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, \quad A \wedge B} \quad (\wedge r)$$

Disjunction rules:

$$\frac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta} \quad (\vee l) \quad \frac{\Gamma \vdash \Delta, \quad A, B}{\Gamma \vdash \Delta \quad , A \vee B} \quad (\vee r)$$

Implication rules:

$$\frac{\Gamma \vdash \Delta, A \quad B, \Gamma \vdash \Delta}{A \to B, \Gamma \vdash \Delta} \quad (\to l) \quad \frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \to B} \quad (\to r)$$

The *weakening rules* allow additional formulas to be inserted on the left or right side. If $\Gamma \vdash \Delta$ holds, then the sequent continues to hold after further assumptions or goals are added:

$$\frac{\Gamma \vdash \Delta}{A, \Gamma \vdash \Delta} \quad (weaken : l) \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A} \quad (weaken : r)$$

Contraction rules allow formula to be used more than once:

$$\frac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta} \quad (contract : l) \quad \frac{\Gamma \vdash \Delta A, A}{\Gamma \vdash \Delta, A} \quad (contract : r)$$

The *cut rule* allows the use of lemmas. Some formula A is proved in the first premise, and assumed in the second premise. The use of the *cut rule* can be removed from any proof, but the proof could get exponentially large:

$$\frac{\Gamma \vdash \Delta, A \quad A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} \quad (cut)$$

The lemma A can be proved from Γ and use A and Γ together to reach the conclusion B.

6 PVS Tactics

PVS is an interactive theorem prover. The interactive theorem prover can have a mechanism to allow users to prove some statement in advance and then reuse the derivation later in the further proofs.

6.1 Derived Rules

If a mechanism for establishing a derived rule is not available, one alternative is to construct a proof script or tactic that can be reapplied whenever a derivation is needed. There are several problems with this. First, it is inefficient instead of applying the derived rule in a single step, the system has to run through the whole proof each time. Second, by looking at a proof script or a tactic code, it may be hard to see what exactly it does, while a derived rule is essentially self-documenting. This suggests a need for a derived rules mechanism that would allow users to derive new inference rules in a system and then use them as if they were primitive rules (i.e. axioms) of the system. Ideally, such mechanism would be general enough not to depend on a particular logical theory being used. Besides being a great abstraction mechanism, derived rules facilitate the proof development by making proofs and partial proofs easily reusable. Also, a derived rule contains some information on how it is supposed to be used and such information can be made available to the system itself [3].

The idea of this approach is using a special higher-order languages for specifying rules, or *sequent schema language*. It means that some logical theory can taken and express its rules using sequent schema. Next step is adding the same languages of the sequent schema to the theory itself.

Overview

This chapter gave the concepts of the theorem prover and the theory behind logic systems. It discussed about the different sorts of the logic and finally a short introduction to PVS tactics and derived rules.

Chapter 5

Type Checking

This chapter describes the PVS type system and the semantic analysis of PVS types. It contains sections on PVS type theory, pretypes, preterms, contexts, and type rules. The PVS prelude is discussed at the end of the chapter. This chapter is based on the descriptions in the formal semantic of PVS, [5].

1 PVS Type System

A type system for a language is a collection of rules for assigning type expressions to various parts of the program. Usually, this can be implemented using a syntax directed definition. An implementation of a type system is called a *type checker*.

The type system of PVS is not algorithmically decidable; theorem proving may be required to establish the type-consistency of a PVS specification. The theorems that need to be proved are called type-correctness conditions (TCCs). TCCs are attached to the internal representation of the theory and displayed on request. There are commands available that attempt to prove the TCCs using built-in prover strategies. The PVS system automatically tracks the status of theories (whether they have been changed, parsed, typechecked etc.) and also takes care of the dependencies among theories. For example, if the specification text of a theory is changed and then a command is issued that requires semantic information, PVS will parse and typecheck the theory automatically. It is often necessary to make changes in theories on which long chains of other theories depend, and frequent reparsing and retypechecking of such theory chains can be very time-consuming. Therefore PVS provides commands which allow limited additions and modifications of declarations without requiring that the associated theories be re-typechecked.

Types serves as a powerful mechanism for detecting semantic errors through typechecking.

- Types impose a useful discipline on the specification.
- Types lead to easy and early detection of a large class of syntactic and semantic errors.
- Types information is useful in mechanized reasoning.

The semantic of PVS will be given by considering a sequence of increasingly expressive fragments of PVS. The semantic of each fragment of PVS will be presented in three steps:

- The first step is to define a set theoretic universe containing enough sets to represent the PVS types.

- The second step is to define a typechecking operation that determines whether a given PVS expression is well typed.
- The third step is to define a semantic function that assigns a representation in the semantic universe to each well-typed PVS type and term.

Type checking checks whether the input program is type-correct. This is a part of the semantic analysis. During type checking, a compiler checks whether the use of names (such as variables, functions, type names) is consistent with their definition in the program. For example, if a variable x has been defined to be of type int, then $x + 1$ is correct since it adds two integers while $x[1]$ is wrong. When the type checker detects an inconsistency, it reports an error. Another example of an inconsistency is calling a function with fewer or more parameters or passing parameters of the wrong type. Consequently, it is necessary to remember declarations so that we can detect inconsistencies and misuses during type checking. This is the task of a *symbol table*. Mainly the symbol table will be used to answer two questions:

- Given a declaration of a name, is there already a declaration of the same name in the current context.
- Given a use of a name, to which declaration does it correspond, or it is undeclared?

Formally, a symbol table maps names into declarations (called attributes), such as mapping the variable name x to its type int. More specifically, a symbol table stores:

- For each type name, its type definition.
- For each variable name, its type.
- For each constant name, its type and value.

PVS is a strongly typed specification language, which means that the compiler can verify that the program will execute without any type errors. Types serves a powerful mechanism for detecting syntactic and semantic errors through typecheckig. The main purpose of type checking are:

- Types impose a useful discipline on the specification.
- Types lead to easy and early detection of a large class of syntactic and semantic errors.
- Types information is useful in mechanized reasoning.

The semantic of higher order logic is given by mapping the well-formed types of the logic to sets, and the well- formed terms of the logic to elements of the sets representing their type. PVS also has predicate subtypes that are to be interpreted over the subsets of the set representing the parent type. The base type in PVS consists of the boolean (`bool`)and the real numbers (`real`). The booleans can be modeled by any two elements set e.g., 2 consisting of the elements 0 and 1, where 0 is empty set and the only element of the set 1.

2 The Type Theory

The simply typed fragment includes types constructed from the base types by the function and product type constructions, and expressions constructed from the constants and variables by means of application, abstraction and tupling. Expressions are checked to be well type under a *context*, which is a partial function that assigns a *kind* (`TYPE, CONSTANT, or VARIABLE`) to each symbol.

2.1 The Simple Type Theory

The PVS specification language is based on simply typed higher-order logic, in which the simple theory of types is augmented by dependent types and predicate subtypes. Type constructors include functions, tuples, records, and abstract data types. There is a large collection of standard theories provided as libraries in PVS. PVS is a strongly typed specification languages. The simply typed fragment include types constructed from the base types by the function and product type constructions and expressions constructed from the constants and variables by means of application, abstraction and tupling. Expressions are checked to be well typed under a context, which is a partial function that assigns a kind (one of `TYPE, CONSTANT, or VARIABLE`) to each symbol, and a type to the constant and variable symbols.

Pretypes

The pretypes of the simple type theory include the base types such as `bool` and `real`. Following are some examples for pretypes: bool, real, [bool, real], [[real, bool]→ bool]. A function pretype from domain pretype A to range pretype B is constructed as $[A \rightarrow B]$. A type is a pretype that has been typechecked in a given context.

Preterms

The preterms of the language consists of the constants, variables, pairs, projections, applications and abstractions. Pairs are of the form (a_1, a_2) where each a_i is a preterm. A pair projection is an expression of the form $p_i\ a$ where $i \in \{1, 2\}$. Abstractions have the form $\lambda(X : T)e$, where X is a variable, T is a type and e is a preterm. Following are some examples for the preterms: TRUE, $\neg TRUE$, λ (x: bool): $\neg X$, p_2 (TRUE, FALSE), (TRUE, $\lambda(x : bool) : \neg(\neg(x))$

Contexts

A context is a sequence of declarations, where each declaration is either a type declaration $s : TYPE$, a constant declaration $c : T$, where T is a type, or variable declaration x : `VAR` T. Preterms and pretypes are typechecked with respect to given context. The empty context is represent as $\{\}$. Following are the well formed rules for contexts:

- A context can also be applied as a partial function so that for a symbol s with declaration D, $(\Gamma, s : D)(s) = D$ and $(\Gamma, s : D(r) = \Gamma(r)$ for $r \neq s$.
- If s is not declared in Γ then $\Gamma(s)$ is undefined.
- If Γ is a context, then for any symbol s the kind of the symbol s in Γ is given by $kind(\Gamma(s))$.
- If the kind of s in Γ is `CONSTANT` or $VARIABLE$, then the $type(\Gamma(s))$ is the type assigned to s in Γ

Example

- bool: TYPE
- TRUE: bool
- FALSE:bool
- x:VAR [[bool, bool]->bool]

Type Rules

The type rules for simple type theory are given by a recursively defined partial function τ that assigns:

- A type $\tau(\Gamma)(a)$ to a preterm a that is well typed with respect to a context Γ.
- The keyword `TYPE` as the result of $\tau(\Gamma)(A)$ when A is a well formed type under context Γ.
- The keyword `CONTEXT` as the result of $\tau(\Gamma)(\Delta)$ when Δ is a well formed context under context Γ. The context Γ is empty for simply typed fragments, so that typechecking is always invoked as $\tau()(\Gamma)$.

The type rules are given by the recursive definition τ.

3 The PVS prelude

The prelude consists of theories that are built in to the PVS system. It is typechecked the same as any other PVS theory, but there are hooks in the typechecker that require most of these theories to be available, hence the order of the theories is important. For example, no formulas may be declared before the booleans are available, as a formula is expected to have type bool. Since definitions implicitly involve both formulas and equality, the booleans theory may not include any definitions. The PVS prelude is a large body of theories that provides the infrastructure for the PVS typechecker and prover, as well as much of the mathematics needed to support specification and verification of systems. The prelude library in PVS [6] is a collection of basic theories about logic, functions, predicates, sets, numbers, and other datatypes. The theories in the prelude library are visible in all PVS contexts, unlike those from other libraries that have to be explicitly imported. During type checking we need to import the meaning of names from the prelude file.

During type checking, we need to define the meaning of the names and variables and to check if they have a right kind or not. If there is no definition for the names, variables, etc, in the theory then it must be found in the imported theory or prelude file. If it is not found during this search then there is an error message. Consider the following example:

Example

Following is an example of a PVS theory and the process of type checking for it. Consider the `"sum"` theory as follow:

```
sum: THEORY
BEGIN
```

```
  n : VAR nat
  sum(n): RECURSIVE nat = ( IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF ) MEASURE (LAMBDA n:n)
  closed_form:THEOREM sum(n) = (n*(n+1))/2
END sum
```

For typechecking the "sum" theory we need to define the meaning of the variables, names etc. For instance the type "nat" for the variable n in the theory is not define, for typechecking the theory, either we have to find the meaning of it in the imported theories or in the prelude file. Furthermore following steps should be performed in order to typecheck the "sum" theory:

- Searching for the meaning of the type "nat" in the "sum" theory.
- If there is no definition and meaning of the type "nat" in the "sum", then we have to look for it in the imported theories.
- Finally if we couldn't find in the imported theories then we have to look for it in the prelude file.

In this example we will import definition of "nat" from the PVS prelude file.

Overview

This chapter covered the main concepts of the PVS type system and semantic analysis in PVS. The main concepts about symbol table and the purpose of the type checker was discussed. In type theory section the concepts: pretypes, preterms and contexts were discussed. In the last section of the chapter, PVS prelude file was discussed.

Chapter 6

Design

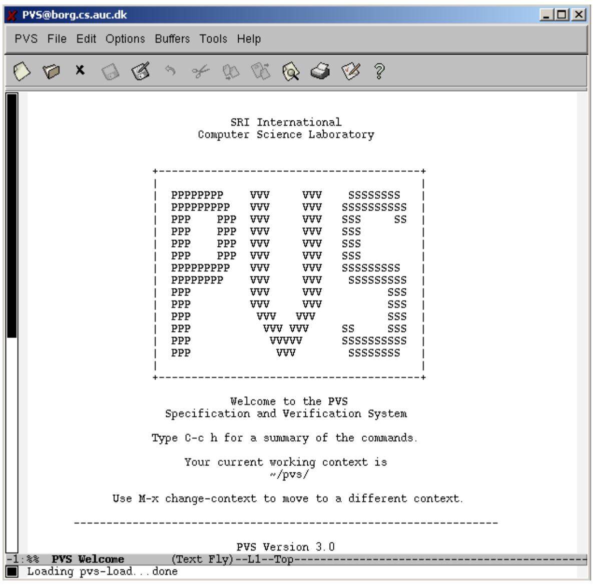
This chapter will describe the design and implementation of PVS system which developed in Java. It describes internal structure of PVS parser, type checker and the PVS graphical user interface which implemented in Jbuilder. The first section of the chapter describes the graphical user interface design. The internal representation of parser which is implemented according to the whole PVS grammar and short introduction of JavaCC which is used for implementing the parser, is describe in the next section. Third section of chapter contains the internal structure of PVS type checker. The last section of the chapter introduce the class for designing the PVS theorem prover.

1 Graphical User Interface for PVS

This section is the describes the two different graphical user interfaces for PVS. The first part describes the user interface created by SRI. Second part contains the description of the user interface created by Jbuilder.

1.1 SRI User Interface for PVS

PVS runs on SUN 4 (SPARC) workstations using Solaris 2 or higher. It has been implemented in common Lisp. It needs a Unix workstation, X-windows, emacs and Tcl/Tk. PVS runs best using the X window system. The Emacs (Gnu Emacs or XEmacs) editors provide the interface to PVS. The user interact with the PVS system through a customized Emacs. Commands can be selected either by pull-down menus or by extended Emacs commands. Extensive help, status-reporting and browsing tools are available, as well as the ability to generate typeset specifications (in user-defined notation) using LaTeX. Editing of specifications is performed with this editor. Instructions are issued to PVS by means of Emacs commands. For example, in order to perform a proof, the cursor is positioned at a formula declaration in the Emacs buffer and the Emacs command M-x prove or the key sequence C-c p is issued. Proof trees and theory hierarchies can be displayed graphically using Tcl/Tk. The PVS interface allows a certain amount of parallel activity. For example, the user can continue editing theories or perform any other activity supported by Emacs while PVS is typechecking a series of theories or performing a lengthy proof. Also, user need not wait for one PVS activity to finish before issuing another command; most commands are queued for execution in the order they were issued, but certain status and other short commands preempt any ongoing analyses, perform their function, and then return the system to its previous activity. Figure 6.1 is the user interface made by SRI:



```
                SRI International
         Computer Science Laboratory

   +----------------------------------------+
   |                                        |
   |  PPPPPPPP    VVV      VVV    SSSSSSSS  |
   |  PPPPPPPPP   VVV      VVV   SSSSSSSSSS |
   |  PPP    PPP  VVV      VVV   SSS      SS |
   |  PPP    PPP  VVV      VVV   SSS        |
   |  PPP    PPP  VVV      VVV   SSS        |
   |  PPP    PPP  VVV      VVV   SSS        |
   |  PPPPPPPPP   VVV      VVV   SSSSSSSSS  |
   |  PPPPPPPP    VVV      VVV    SSSSSSSSS |
   |  PPP         VVV      VVV          SSS |
   |  PPP         VVV      VVV          SSS |
   |  PPP          VVV    VVV           SSS |
   |  PPP           VVV VVV      SS     SSS |
   |  PPP            VVVVV       SSSSSSSSSS |
   |  PPP             VVV         SSSSSSSS  |
   |                                        |
   +----------------------------------------+

              Welcome to the PVS
      Specification and Verification System

      Type C-c h for a summary of the commands.

           Your current working context is
                      ~/pvs/

    Use M-x change-context to move to a different context.

  -----------------------------------------------------------------

                   PVS Version 3.0
-1:%%  PVS Welcome      (Text Fly)--L1--Top------------------------
Loading pvs-load...done
```

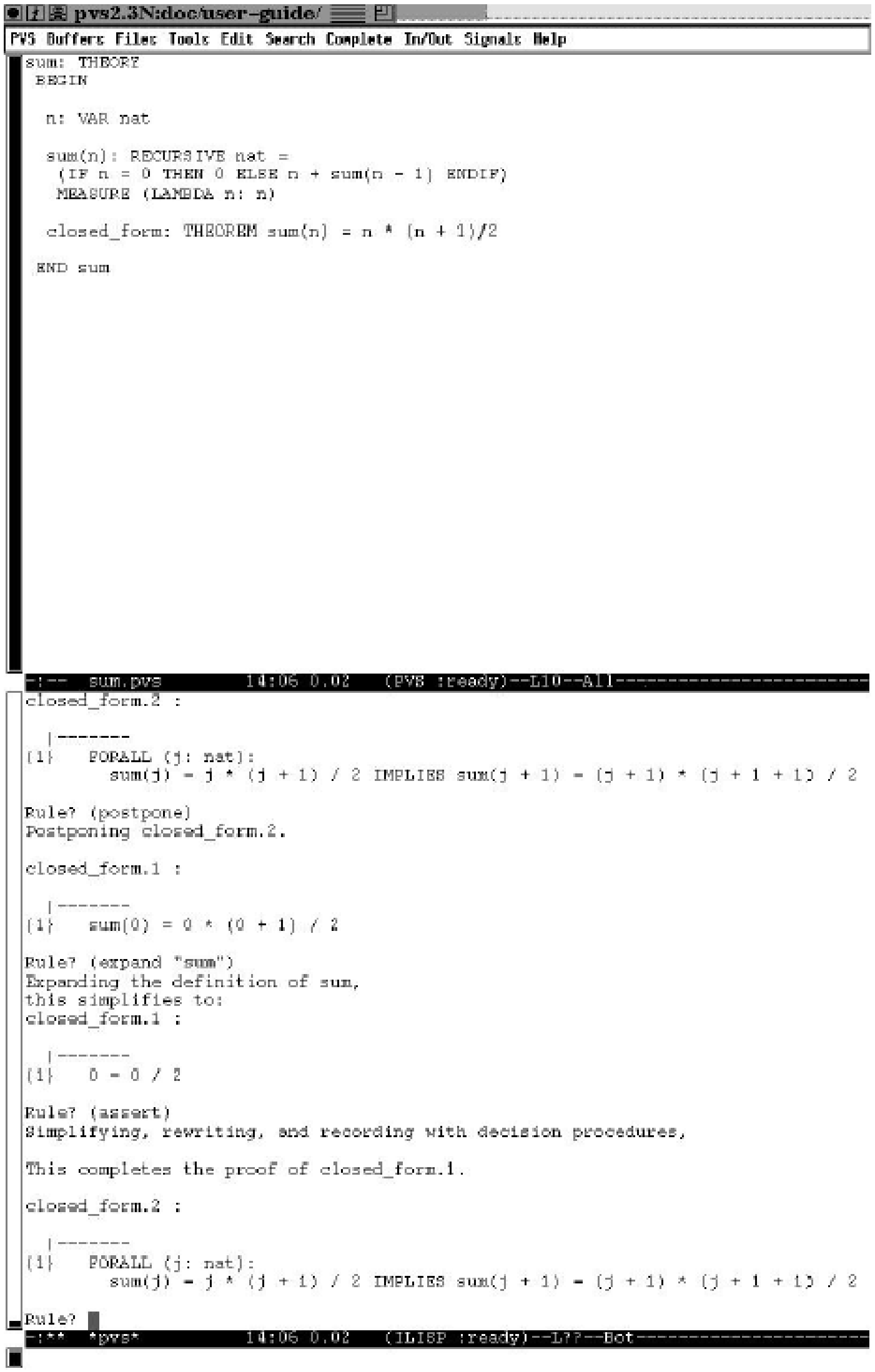
Figure 6.1: User interface for original PVS from SRI

Create The Specification in SRI User Interface

The figure 6.2 shows a specification for summation of the first n natural numbers, as it appears in emacs. The `sum` specification is in the top window, and a proof is in progress in the bottom. The mode line indicates that PVS is ready for a command. As mentioned in chapter two (the PVS architecture), there are two actors in the PVS use case (see figure 2.1), : theory designer and verifier. When the verifier starts to prove, a graphical representation of PVS proof tree will show up in the new window. This will show the tree representation of the proof and proof steps. As mentioned before the basic objective of the proof is to generate a proof tree of sequent, while proving the PVS will always be looking at unproved leaf of the tree, which is call *current* sequent. When a given branch is complete (e.g., ends in a proved leaf), the prover automatically moves on to the next unproved branch, or, if there are no more unproven branches, then the prove is complete. All the proof steps and proof trees will show in the figure 6.3. This figure is the proof steps for proving the `sum` theory. It covers all the PVS rules which has been used for proving the `sum` theory.

1.2 User Interface Created in JBuilder

The user interface created in JBuilder is a simple text editor, capable of reading, writing and editing the selected theory. This text editor will be able to set the text color, as well as the background color of the text editing region in the `edit` menu. The `action menu` of the user interface, include the parsing, type checking and proving of the selected theory. `Help` menu contain information about the user interface and a link to PVS prover guide. Status-reporting are also available. The two actors in PVS: theory designer and verifier,


pvs2.3N:doc/user-guide/
PVS Buffers Files Tools Edit Search Complete In/Out Signals Help

```
sum: THEORY
 BEGIN

  n: VAR nat

  sum(n): RECURSIVE nat =
   (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
   MEASURE (LAMBDA n: n)

  closed_form: THEOREM sum(n) = n * (n + 1)/2

 END sum

-:--  sum.pvs        14:06 0.02   (PVS :ready)--L10--All-----------------------
closed_form.2 :

  |-------
{1}   FORALL (j: nat):
        sum(j) = j * (j + 1) / 2 IMPLIES sum(j + 1) = (j + 1) * (j + 1 + 1) / 2

Rule? (postpone)
Postponing closed_form.2.

closed_form.1 :

  |-------
{1}   sum(0) = 0 * (0 + 1) / 2

Rule? (expand "sum")
Expanding the definition of sum,
this simplifies to:
closed_form.1 :

  |-------
{1}   0 = 0 / 2

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed_form.1.

closed_form.2 :

  |-------
{1}   FORALL (j: nat):
        sum(j) = j * (j + 1) / 2 IMPLIES sum(j + 1) = (j + 1) * (j + 1 + 1) / 2

Rule?
-:**  *pvs*          14:06 0.02   (ILISP :ready)--L??--Bot---------------------
```

Figure 6.2: The sum specification in PVS generated by SRI

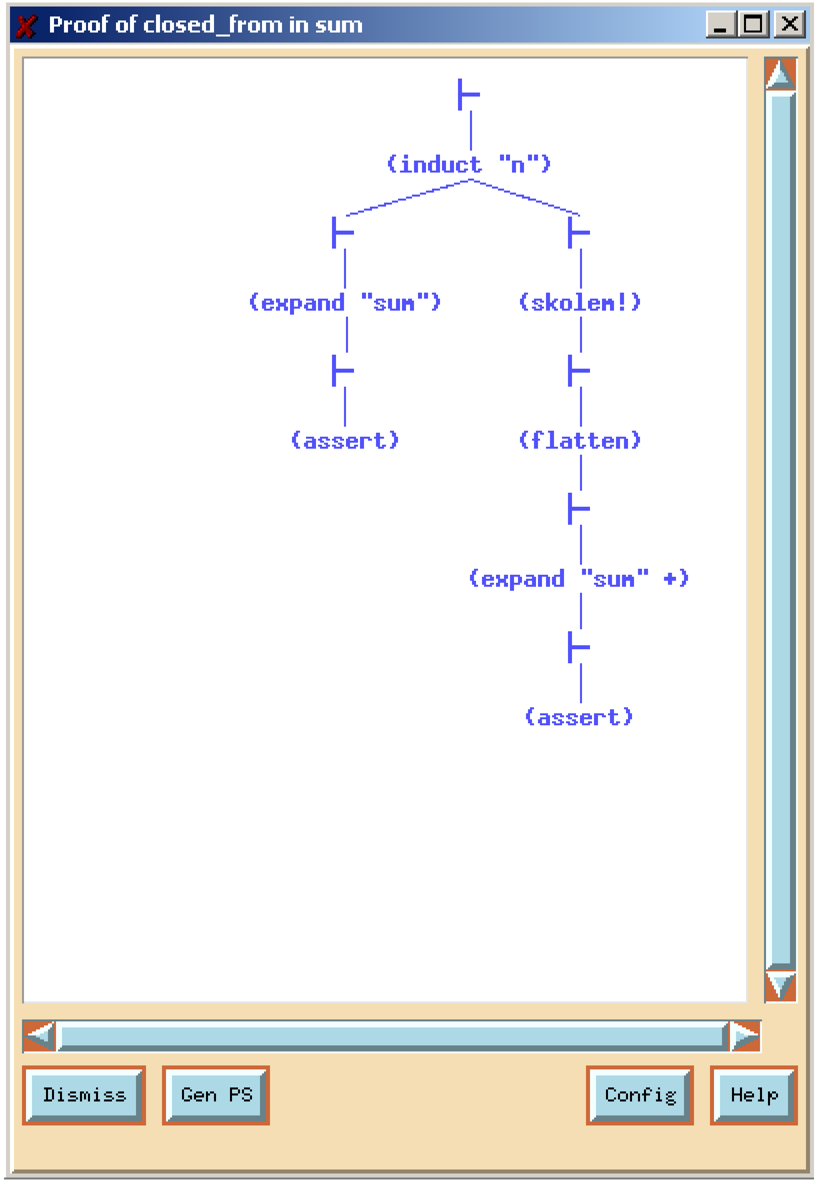


Figure 6.3: The Graphical Iterface of the PVS Created by SRI

have two different parts in this user interface. Following is the description of the actors work in the graphical user interface, created in Jbuilder.

1.3 User Interface For Theory Designer

Theory designer can either start typing the desire theory in the text editor area or can select one theory from a file. The font of the selected theory can be edit via edit menu. The background color of the text in the text area can also be edited in the edit menu. The text area in the text editor has syntax highlight. The PVS reserve words, symbols and identifiers have different colors in this text area. Selected theory can be parse and type check in the action menu. The theory can be proved after parsing and type checking. Following are the steps and the description of making a user interface for theory designer in JBuilder.

Setting the look and feel

The runtime look and feel is determined by the source code setting. The look and feel selected in JBuilder's designer is for preview purposes and has no effect on the source code. Look and feel can be changed in JBuilder. There are several choices of look and feel available in JBuilder:

- Metal.
- CDE/Motif.
- Window.
- MacOS Adaptive (supported only on Macintosh platforms).

In this project *metal* look and feel has been selected. To change the look and feel we have to set up Jbuilder so the designer can use the *metal* look and feel. By setting up the look and feel in the designer context menu or in the Jbuilder options dialog box, it doesn't have any effect at runtime. To force a particular runtime look and feel, we have to set the look and feel explicitly in the `main ()` method of the class that runs the application. By default, the application wizard generates the following line of the code in the `main()` method of the runnable class:

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

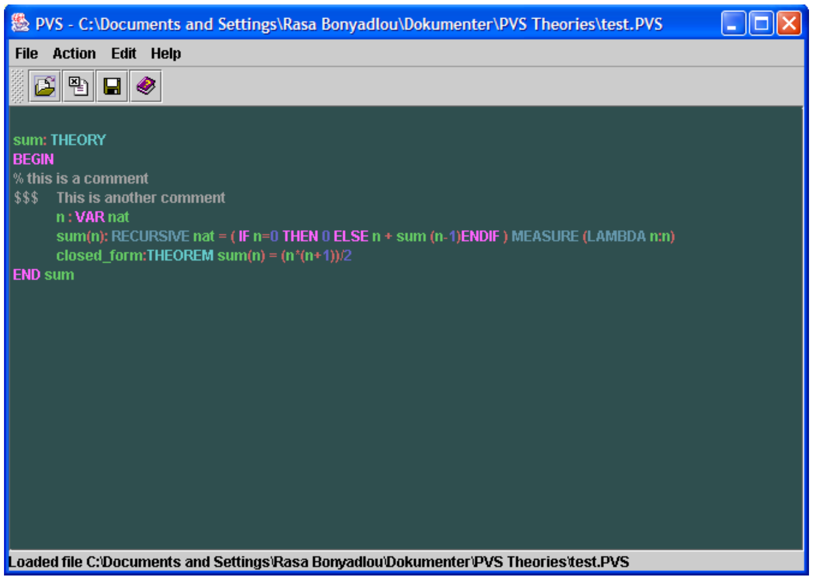
That means the runtime look and feel will be whatever the hosting system is doing. To specify metal, the code has to be changed as follow:

```
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
```

After saving the project and its files, we can run the application and the metal look and feel will appear in the user interface.

Adding The Text Area

In this step we add the text area which completely fill user interface frame between the menu bar at the top and the status bar at the bottom. To support this, the layout manager for the main user interface container needs to use BorderLayout. For this purpose, a panel containing multiple components is considered as one component. A north component clings to the top of the container, an east component to the left side, and so on. A component placed into the center area completely fills the container space not occupied by any other areas containing components. The application wizard creates a JFrame component that's the main container for this UI. The figure 6.4 represent the user interface of the PVS created in JBuilder in this step. The user can either enter the desirer theory in the text editor or choose one of the theories by selecting File|Open.


PVS - C:\Documents and Settings\Rasa Bonyadlou\Dokumenter\PVS Theories\test.PVS
File
Action
Edit
Help

```
sum: THEORY
BEGIN
% this is a comment
$$$    This is another comment
        n : VAR nat
        sum(n): RECURSIVE nat = ( IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF ) MEASURE (LAMBDA n:n)
        closed_form:THEOREM sum(n) = (n*(n+1))/2
END sum
```

Loaded file C:\Documents and Settings\Rasa Bonyadlou\Dokumenter\PVS Theories\test.PVS

Figure 6.4: The User Interface of PVS Created by JBuilder

Create The Menus

In this step the following menus are created:

- File
- Action
- Edit
- Help

Part of the menus is shown in figure 6.5. The menu designer is used to create and edit the menus. We can create new menu items, add a new menu and insert a separator bar.

Setup The Tool Bars

By selecting toolbar option in the application wizard, JBuilder generates code for a JToolBar and populates it with three Jbutton components that already display icons. The text for each buttons label must be specified. The buttons in this user interface are, open file which allows the user to open and selected one

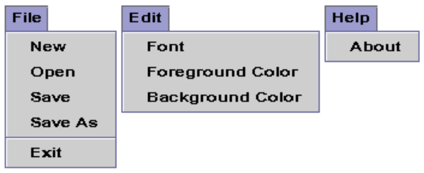


Figure 6.5: The menus

PVS theory from a file. Clean, which can clean the text area, save for saving theories in the file and the help button which is a link to PVS home page generated by SRI.

Adding a right-click menu to the text area

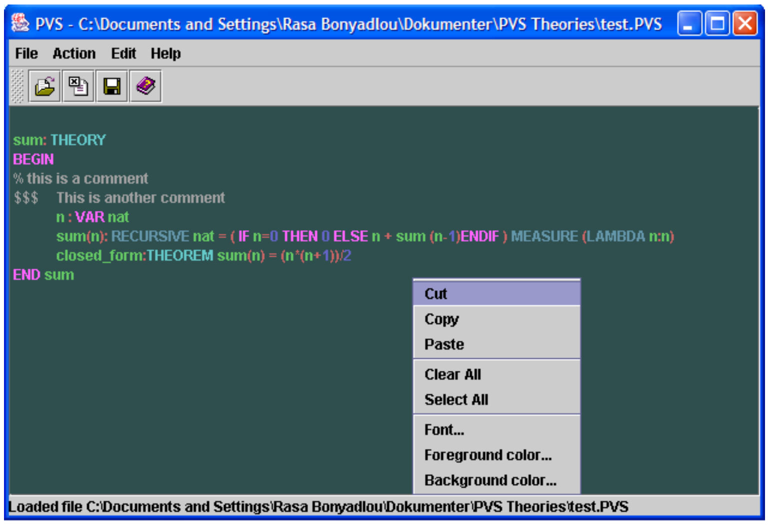
The DBTextDataBinder component adds a right-click menu to Swing text components for performing simple editing tasks such as cut, copy, paste, clear all, select all, undo and redo clipboard data. DBTextDataBinder also has built-in actions to load and save files into a JTextArea, but it don't allow file name to be retrieved. This step is the description of adding the DBTextDataBinder, binding it to the JTextArea1 and suppress the file open and save actions. Following are the steps for adding a right-click menu to the text area:

- In the design tab the DBTextDataBinder component on the dbSwing models tab of the palette is be selected.
- In can be dropped anywhere in the designer or on the component tree.
- dBTextDataBinder1 in the component tree, is selected and jTextComponent property in the inspector is change to jTextArea1 from the drop-down list. This binds dBTextDataBinder1 to jTextArea1 by placing the following line of code in the jbInit() method.

  ```
  dBTextDataBinder1.setJTextComponent(jTextArea1);
  ```

- The enableFileLoading property for dBTextDataBinder1 is selected and its value is set to false by using the drop-down arrow.

By saving the changes and running the application a pop up menu will appear when the user right-clicks the text area. Figure 6.6 shows the graphical user interface created in JBuilder.



```
sum: THEORY
BEGIN
% this is a comment
$$$    This is another comment
        n : VAR nat
        sum(n): RECURSIVE nat = ( IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF ) MEASURE (LAMBDA n:n)
        closed_form:THEOREM sum(n) = (n*(n+1))/2
END sum
```

```
Loaded file C:\Documents and Settings\Rasa Bonyadlou\Dokumenter\PVS Theories\test.PVS
```

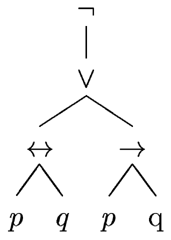
Figure 6.6: Right Click Menu

1.4 User Interface For Verifier

In this part the verifier is started to prove the written theory by using the PVS proof rules. After parsing and type checking, the selected theory in the text area must be proved, by selecting the proof part in the action menu.

Internal Representation Of PVS Parser

The PVS parser builds a parse tree according to the PVS grammar. The act of *parsing* is to determine the grammatical structure of a sentence of natural languages. For propositional logic the most obvious way of parsing a formula is to turn the string into a tree, as illustrated by the following example: Let ϕ be the formula $\neg((p \leftrightarrow q) \vee (p \rightarrow q))$ this formula can parse ϕ into a tree as :



A tree is an ordered pair: the first item is the label of the tree root and the second item is a list of all the children of the root. Each child is itself a tree. This is a highly recursive definition and trees have recursive structures. For each formula ϕ there is a tree formula $\hat{\phi}$ that represents the tree corresponding to ϕ. Below is the definition of the three kinds of tree formulas corresponding to the tree kinds of the formulas:

- For a single proposition p, the tree consists of a single node labeled p which has no children. Thus $\hat{p} = [p, []]$, the first argument of the pair is the label of the root and the second argument is the list of the children of the root, which is an empty list in this case.

- For a negated formula ,$\neg\phi$, the root of the tree is labeled by $\neg$ and the root has exactly one child. This child is the tree representation of ϕ or $\hat{\phi}$. Thus: $\widehat{\neg\phi} = [\neg, [\hat{\phi}]]$.

- Finally for a formula of the form $\phi \circ \psi$ where $\circ$ is a binary connective (either $\wedge$, $\vee$, $\rightarrow$, or $\leftrightarrow$) the root of the tree is labeled by $\circ$ and the root has two children, the first child is $\hat{\phi}$ and the second is $\hat{\psi}$. Thus: $\widehat{(\phi \circ \psi)} = [\circ, [\hat{\phi}, \hat{\phi}]]$.

The programming language used for implementation of PVS parser is JavaCC. JavaCC stands for Java Compiler Compiler and it is a parser generator and lexical analyzer generator. It can read description of a language and generate the code.

2 PVS Parse Tree

The parser transforms an input stream into a parse tree. The parse tree generated by a parser made with JavaCC, is the LLR parse tree. Furthermore it is a Look-ahead LR parser, (LR means left-to-right signifying a parser which reads the input string from left to right), look-ahead helps in knowing if the complete rule has been matched or not.

2.1 Look ahead

The job of a parser is to read an input stream and determine whether or not the input stream conforms to the grammar. This determination in its most general form can be quite time consuming. The general problem of matching an input with a grammar may result in large amounts of backtracking and making new choices and this can consume a lot of time. The amount of time taken can also be a function of how the grammar is written. Many grammars can be written to cover the same set of inputs or the same languages (e.g., there can be multiple equivalent grammars for the same input language). The performance hit from such backtracking is unacceptable for most systems that include a parser. Hence most parsers do not backtrack in this general manner (or do not backtrack at all), rather they make decisions at choice points based on limited information and then commit to it [4].

Parsers generated by Java Compiler Compiler (JavaCC) make decisions at choice points based on some exploration of tokens further ahead in the input stream, and once they make such a decision, they commit to it. The process of exploring tokens further in the input stream is termed "looking ahead" into the input stream. There are 4 different kinds of choice points in JavaCC:

- An expansion of the form: $(exp1|exp2|...)$. In this case, the generated parser has to somehow determine which of $exp1$, $exp2$, etc. to select or continue parsing.

- An expansion of the form: $(exp)?$. In this case, the generated parser must somehow determine whether to choose exp or to continue beyond the $(exp)?$, without choosing exp. Note: $(exp)?$ may also be written as $[exp]$.

- An expansion of the form $(exp)*$. In this case, the generated parser must do the same thing as in the previous case, and furthermore, after each time a successful match of *exp* (if *exp* was chosen) is completed, this choice determination must be made again.
- An expansion of the form $(exp)+$. This is essentially similar to the previous case with a mandatory first match to exp.

There is an abstract form of PVS parse tree in the figure 6.7, generated by JavaCC lexical analyzer by applying the PVS grammar.

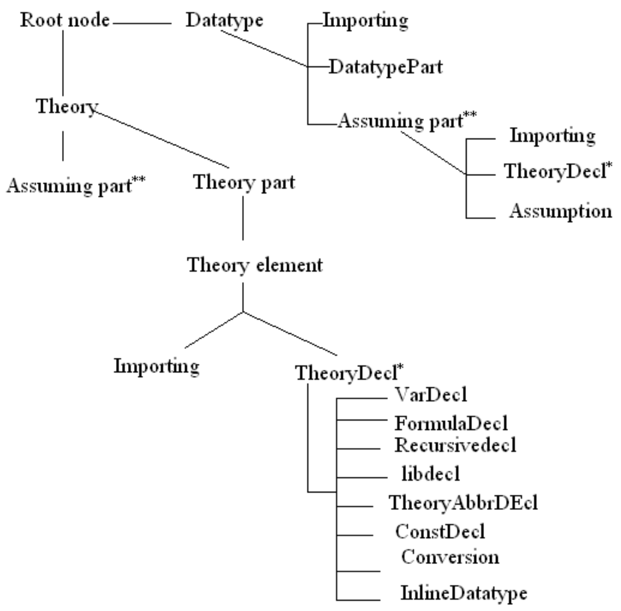


Figure 6.7: PVS Parse Tree

This is an abstract representation of the PVS parse tree, which is obtained by PVS grammar (Appendix A).

3 Type Checker

The PVS typechecker analyzes theories for semantic consistency and adds semantic information to the internal representation built by the parser. In order to implement the PVS type checker a symbol table require, which holds all the information about symbols needed during typechecking. The *semantic analysis* phase of a compiler, checks that each expression has a correct type, and translate the abstract syntax into a simpler representation suitable for generating the code.

Symbol Table

Typechecking is characterized by maintenance of *symbol tables* mapping identifiers to types and values. As the declarations of types, variables and constants are processed, these identifiers are bound to meanings

in the symbol tables. According to the PVS languages references [8], *local* declaration for variables may be given, in association with constant and recursive declarations and *binding expressions* (e.g., involving `FORALL` or `LAMBDA`). Each local variable in a program has a scope in which it is visible. For example, let D in E end, all the variables, types and functions declared in D are visible only until the end of E. As the semantic analysis reaches the end of each scope, the identifier local to that scope are discarded. A symbol table is a set of *bindings* denoted by the $\mapsto$ arrows. For example the symbol table σ_0 contains the bindings $g \mapsto string, a \mapsto int$; means that the identifier a is an integer variable and g is a string variable. PVS is able to do several activities at the same time, for example saving proof and typechecking another theorem can be done at the same time. Each module or part in PVS has a symbol table σ of its own. Some PVS theories can be very large and they may contain thousands of identifiers, symbol tables must permit efficient lookup. For this purpose a hash table is associated.
The symbol table is a vector, consisting of objects of the type symbol, each symbol has an id and a kind. The kind defines whether it is a VAR, CONST, TYPE, etc. Id contains a string which represent the individual symbols. Following is an example of PVS theory and symbol table for it:

```
sum: THEORY
 BEGIN
  n : VAR nat
   sum(n): RECURSIVE nat = ( IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF ) MEASURE (LAMBDA n:n)
     closed_form:THEOREM sum(n) = (n*(n+1))/2
END sum
```

The symbol table for the "sum" theory will be:

Id	Kind
n	VAR
nat	UNDEFINED
sum	CONST
closed-form	FORMULA

When the "prelude" is loaded, the kind of `nat` is found to be `TYPE`.

3.1 Internal Structure of Type Checking

According to the PVS parse tree, figure 6.7, which is generated by JavaCC by applying the whole PVS grammar (Appendix A), we have two main sub nodes: *theory* and *datatype*. In order to implement the PVS type checker, we ignore the *datatype* part and start from *theory part*. There are three type checking methods which will developed:

- Compile
- Type Checking
- Save

Compile and type checking methods are implemented in the Tree root part. The saving method is implemented in the theories part. Following three sections are the internal structure of these three methods:

3.2 Compile

The compile method creates a data structure for checking correct use of variables, etc. It, compiles all theories (NodeTheory) in this parse tree. Mainly the compile method generates a symbol table for checking the declaration parts. The symbol table is a data structure for storing and retrieving information about variables, constants, types etc. IdTable contains pairs (Identifier (a string) and a index (integer) identifying the position of the symbol in the symbol table with the given identifier. The compile method checks for correct declaration of variables, types, etc. For example it checks for duplicate declaration of variables.
A symbol table is created for each theory. The structure consists of `Theories` objects which are parsed to the compile method of the tree root. The tree root in turn calls the compile() method of the individual `NodeTheory` objects. The compile method of the `NodeTheory` objects constructs `Theory` objects and adds these to the theories object. Furthermore the compile method of any sub nodes is called, the sub nodes of these is in turn called etc. When the compile function of a declaration node is called, this nodes compile method adds a corresponding symbol to the relevant `Theory` object symbol table, if the symbol has already been declared an error is thrown.
Following is the internal structure of the compile method which is in the tree root. According to the UML representation of the tree root, figure 6.8:

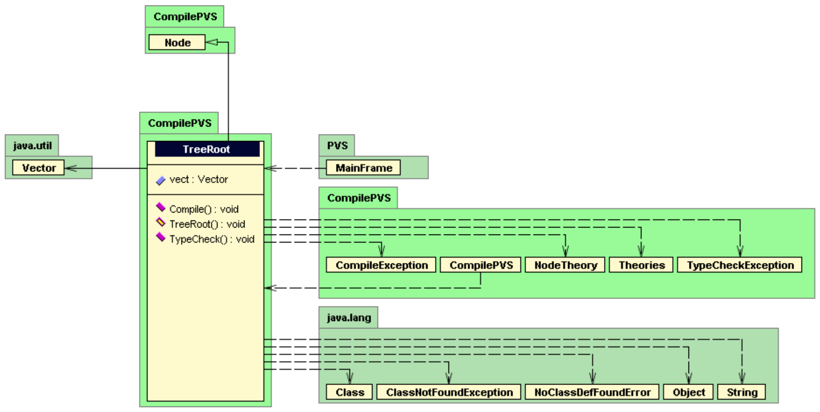


Figure 6.8: TreeRoot UML Representation

There are two methods in the TreeRoot class: compile() and typeCheck().
During compilation, when we meet a name, it can be in two contexts:

- Defining
- Using

Defining

That is when we want to introduce a new name for something in the theory. The process will be as follows:

- Lookup the name in the IdTable, if it occurs already, then there is an error, because the name is used in two senses.

- Create entries for the name in the IdTable and in the Symbols. The defining context gives the kind.

Using

When we need the meaning of a name. The procedure will be as follows:

- Lookup the name, if it occurs, it is defined in this theory and we have its meaning and we can check whether it is the right kind or not.
- If it is not found in the current theory it must be found in an imported theory or the prelude. So we have to search these theories in the reverse order of importing. Imported theories are loaded into the Theories table when IMPORTING is processed, the prelude is loaded by default before any type checking takes place.
- If it is not found during this search, we have an error (undefined name).
- If it is found, we have to import it. We create entries for it in the identifier table and symbol table - copying its meaning from its definition. And then we proceed as in first part (see following example).

Example

Consider the "sum" theory as follows:

```
sum: THEORY
 BEGIN
  n : VAR nat
   sum(n): RECURSIVE nat = ( IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF ) MEASURE (LAMBDA n:n)
    closed_form:THEOREM sum(n) = (n*(n+1))/2
END sum
```

In this theory we will import "nat" from the PVS prelude file.

3.3 TypeCheck Method

This method, checks for correct use of variables, types etc. The `thypeCheck` method calls typecheck on all theories (`NodeTheory`) in the parse tree. The TypeCheck method goes through the parse tree and for each theory it checks if the individual variables type, is correct. This method starts in the TreeRoot class (see figure 6.8).

3.4 Saving Method

This method saves the compiled files with ".cpt" extension. PVS theories will be saved with "cpt" extension after type checking. The saved ".cpt" file contains all the theories and symbol tables of the compiled PVS file.
Theories imported in PVS using "IMPORTING xxx" refers to compiled theory files (containing one or more theories) rather than individual theory names. The PVS files may contain more than one theory e.g.,

prelude.pvs which contains 89 theories. As a result of this the save method has been modified so that all the theories in a pvs file will be saved in a single cpt file with the same name, e.g., compiling prelude.pvs outputs prelude.cpt instead of 89 separate compiled theories. As an example of "cpt" saved files, consider the following `sum` theory:

```
sum: THEORY
 BEGIN
     n : VAR nat
     sum(n): RECURSIVE nat = ( IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF ) MEASURE
     (LAMBDA n:n)
     closed_form:THEOREM sum(n) = (n*(n+1))/2
 END sum
```

The saved "cpt" file for `sum` theory will be as follows:

```
//---------------------------------------//
//            COMPILED PVS FILE          //
//---------------------------------------//
THEORY "sum"
TYPECHECK "true"
SYMBOLS 3
//---------------------------------------//
4 "n"
6 "sum"
5 "closed_form"
//---------------------------------------//
```

The following three sections will outline the internal structure of class *Theories*, *Theory* and *Symbol*.

3.5 Class Theories

The theories class represents the whole complied PVS project. It contains a vector of the theories, which is each represented by a theory object. The save method, is used in order to save the compiled PVS project as CPT files. Figure 6.9 is the UML representing of the class theories:

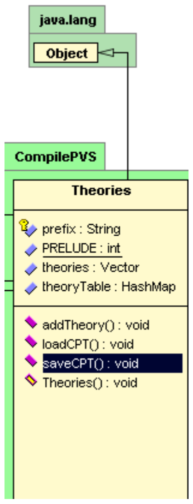


Figure 6.9: UML Representation for Class Theories

Following are the methods in the class theories:

- addTheory
- loadCPT: loads all complied theories in a CPT file to theories list.
- saveCPT: saves compiled PVS project as "cpt" files.

3.6 Theory

This class represent the individual compiled theories in the PVS file. It also contains symbol table, name of the theory and origin variable. Origin is the variable that can be either user/imported or prelude. This is used in order to keep track of where the individual theories origin. So that when a CPT file is saved only theories marked USER are saved and not PRELUDE and IMPORTED theories. A theory can be saved in a file with "to Stream" method and it can be loaded by "fromStream". Figure 6.10 is the UML representation of the class `Theory`:

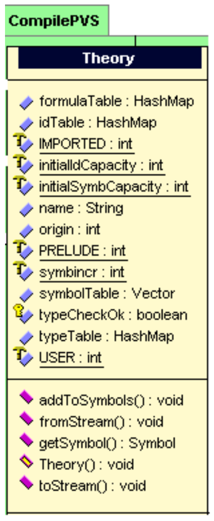


Figure 6.10: UML Representation for Class Theory

This class contains four methods:

- addToSymbolTable: adds a symbol object to the symbol table and creates a link from its id in the id table to the index in the symbol table.
- toStream: saves theories to file.
- fromStream: it loads theories from "cpt" files.

3.7 Symbol

This class is super class for the entities in the symbol table. Figure 6.11 is the UML representing of the class Symbol:

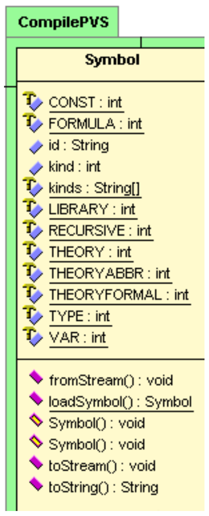


Figure 6.11: UML Representation for Class Symbol

This class contain following methods:

- fromStream
- loadSymbol: loads a symbol from file and creates a corresponding new symbol object which is returns.
- toStream: saves a symbol to a file.
- toString

3.8 Internal Representation of Type Checker

Following is a representation of the upper level nodes possible in a parse tree, which regards the to the TypeCheck() method. The tabulation of the individual node names represent their level in the parse tree. All possible branches of a parse tree either ends in a part unimportant to the TypeCheck() method or ends up in either a NodeExpr or a NodeTypeExpr. These two nodes and any nodes below their level in the parse tree, are omitted. The main part of the type checking takes part in the NodeTypeExpr and NodeExpr. Declarations has sub node of the type NodeTypeExpr, and as such this node is important to the type checking.

```
TreeRoot
    NodeTheory
        NodeTheoryFormals
        NodeTheoryFormal
            NodeTheoryFormalDecl
                NodeTheoryFormalType
                NodeTheoryFormalConst
                NodeTypeExpr *
            NodeImporting
        NodeExporting
            NodeExportingNames *
            NodeExportingTheories
            NodeExportingName
        NodeAssumingPart
            NodeAssumingElement
                NodeImporting
        NodeTheoryDecl
            NodeAssumption *
        NodeTheortPart
            NodeTheoryElement
                NodeImporting
                NodeTheoryDecl
                NodeVarDeacl
                NodeFormulaDecl
                NodeRecursiveDecl
                NodeLibDecl **
                NodeTheoryAbbrDecl **
                NodeConstDecl
                NodeTypeDecl
                NodeInductiveDecl **
                NodeConversion
                NodeInlineDatatype **
                NodeJudgement
                NodeAutoRecursiveDecl **
```

NodeConversionEllem
NodesuptypeJudgment *
NodeConstantJudgment *
NodeRewritenam
NodeTypeExpr
    NodeTypeApplication *
    NodeName
    NodeEnumerationType *
    NodeSuptype *
    NodeFunctionType *
    NodeTupleType *
    NodeRecordType *

In this tabulation of individual node, the nodes with * are not implemented and the nodes with ** are not supported in this version of PVS. The type checking method is done for the rest of the nodes.

4 Theorem Prover

As describe in chapter two, the structure of PVS proofs consists of a proof tree and the goal of the proofs is to construct a proof tree. If all the leaves are recognize as true, then the proof tree is complete. Each proof goal is a *sequent* consisting of a sequence of formulas called *antecedents* and a sequence of formulas called *consequents*. Following is internal representation of sequents:

```
public class Sequent {

public Vector  antecedents, consequents; // of Expression
// hidden expressions among the antecedents and consequents
public BitSet  ahidden, bhidden;
```

The default constructor sequent creates an empty sequent.

```
public Sequent() {
// creates new empty sequent
antecedents = new Vector();
consequents = new Vector();
ahidden = new BitSet();
chidden = new BitSet();
};
```

The constructor sequent makes a copy of an existing sequent.

```
public Sequent(Sequent s) {
// creates a copy of an existing sequent
antecedents = new Vector(s.antecedents);
consequents = new Vector(s.consequents);
ahidden = s.ahidden.clone();
chidden = s.bhidden.clone();
};
```

The following two methods: *addConsequent* and *addAntecedent* add to the end of the vectors.

```
public void addConsequent(Expression e) {
consequents.add(e);
}



public void addAntecedent(Expression e) {
antecedents.add(e);
}
```

Following method hides a formula by setting a corresponding bit:

```
public void hide(int i) {
// hides an expression
if (i<0 && i >= -consequents.size) chidden.set(-1-i);
else if (i > 0 && i <= antecedents.size) ahidden.set(-1+i);
}

//*
public void delete(int i) {
// deletes an expression
if (i<0 && i >= -antecedents.size) {
for (int k = -i; k < ahidden.length(); ++k)
if (ahidden.get(k+1)) ahidden.set(k); else ahidden.clear(k);
antecedents.removeElementAt(-1-i);
} else if (i > 0 && i <= consequents.size) {
for (int k = -i; k < ahidden.length(); ++k)
if (chidden.get(k+1)) ahidden.set(k); else chidden.clear(k);
consequents.removeElementAt(-1+i);
}
}
//*

public boolean check() {
// checks whether this sequent is a tautology
for (int i = antecedents.size-1; i >= 0; --i) {
// check whether antecedent(i) occurs as a consequent
Expression a = antecedents.get(i);
for (int k = consequent.size-1; k >= 0; --k)
if (consequents.get(k).equal(a)) return true;
}
return antecedents.size  == 0;
}
}
```

Proof Tree

PVS proof steps builds a proof tree by adding subtrees to leaf nodes as directed by the proof commands. Following is the class which can be used to make a proof tree:

```
public class ProofStep {

public ProofStep  parent
public Vector children // of ProofStep

public Sequent seq  // the sequent corresponding to this proof step

public ProofStep(Sequent s) {
// creates new initial proof step
seq = s;
parent = this; // no parent
children = new Vector(); // no children
};
public ProofStep(Sequent s, ProofStep parent) {
// adds this as a child to the parent
seq = s;
this.parent = parent;
parent.children.add(this);
};
public void delete() {
// deletes the children steps
children.clear();
}
public boolean check() {
// checks whether this proof step is completed
boolean ok  = true;
if (children.size > 0)
// the completion is the conjunction of completed children
for (int k = children.size-1; k >= 0 && ok; --k)
ok = ok && children.get(k).check();
else ok = seq.check();
}
}
```

Overview

This chapter covered the internal structure of different parts of the PVS system, which was developed in JBuilder. The first section covered the design details for making PVS GUI in JBuilder. The two different user interface for the PVS system were discussed: the first part was the GUI of the original PVS which is distributed by SRI and the second part was the design details for the user interface which was generated in Jbuilder. The next section covered the internal representation of the PVS parser and parse tree. Design of the PVS type checking was discussed in the third section. The different classes and methods in type checker design, were also discussed. The last section of the chapter introduced the classes for implementation of a PVS prover.

Chapter 7

Test

This chapter will describe the testing instructions and outputs for the PVS system which were developed using Java language. The first section will contain the test instructions and the output for the PVS parser, which is generated by JavaCC. The next section contains the test instructions and cases for PVS graphical user interface, (generated by JBuilder. Last part describes the test cases and instructions for the PVS typechecker. Test cases for each part are examples of PVS theories.

1 Parser Testing

The parser checks for syntactic errors in the given theory. PVS parser generates parse tree according to the full PVS grammar (Appendix A). The parser was generated by applying the PVS grammar (Appendix A) to JavaCC. PVS user can either type an arbitrary PVS theory or selected one from the open menu. Selected theory can be parse by selecting the parse from action menu in GUI. Following are the test requirements, instructions and output for the parser.

1.1 Test Requirements

PVS parser checks for the syntactic errors in the given theories . The parser for PVS grammar is a program which takes the language string as its input and produces either a corresponding parse tree or an error. The PVS parser was generated by applying the PVS grammar to JavaCC. JavaCC is a parser generator for use with Java application, which is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. The PVS parser is tested for correct output. Correct output being the whole (input) theorem apart of the commands which are irrelevant to the theory/theories.

1.2 Test Cases Specification

The two main approaches to testing software are: "black-box" (or functional) testing, and "white-box" (or structural) testing. For testing the PVS parser, white box strategies has been used. White-box testing strategies include designing tests such that every source line of code is executed at least once, or requiring every function to be individually tested. It focuses specifically on using internal structure of the PVS parse tree to guide the selection of test data (PVS theories). The goal of the testing PVS parser is to obtain the correct output. Correct output being the whole (input) theorem apart of the commands which are irrelevant to the theory/theories.

Following are selected PVS theories as a test cases for the parser. According to the PVS grammar (Appendix A), PVS theories contain specification, importing and exporting, assuming part, theory part, declarations, type expressions, expressions, names, identifiers and datatype parts. Since we apply white box strategies to the PVS parse tree, selected test cases should contain specification, importing and exporting, assuming part, theory part, declarations, type expressions, expressions, names, identifiers and datatype parts (in order to test every function individually). The PVS theories may not include all of these parts at once. Furthermore a PVS theory may or may not contain specification, importing and exporting, assuming part, theory part, declarations, type expressions, expressions, names, identifiers and datatype parts.

1.3 Test Cases

Bellow are the three PVS theories to which the PVS parser is applied. The first theory is the `sum` theory which is discussed in the proving part.

1.4 Test1: Sum

```
sum: THEORY
BEGIN
% this is a comment
        n : VAR nat
        sum(n): RECURSIVE nat = ( IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF ) MEASURE
 (LAMBDA n:n)
        closed_form:THEOREM sum(n) = (n*(n+1))/2
END sum
```

After generating parse tree the output will be the same as the theory except the comments:

```
sum : THEORY
BEGIN
        n : VAR nat;
        sum( n ) : RECURSIVE nat = (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF )
 MEASURE ( LAMBDA n:n );
        closed_form : THEOREM sum(n) = (n * (n + 1)) / 2;
END sum
```

1.5 Test2: Square-root

Here is another PVS theory:

```
square_root.pvs

%  Defines square root
%  Specializes the lemmas on roots


square_root : THEORY

  BEGIN

  IMPORTING roots

  x, y : VAR nonneg_real

  px, py : VAR posreal

  u, v : VAR real

  --------------
    Definition
  --------------

  sqrt(x) : nonneg_real = root(x, 2)

  sqrt_def : LEMMA sqrt(x) = y IFF y * y = x

  square_sqrt : LEMMA  sqrt(x) * sqrt(x) = x

  sqrt_square : LEMMA sqrt(x * x) = x

  square_sqrt2 : LEMMA  expt(sqrt(x), 2) = x

  sqrt_square2 : LEMMA sqrt(expt(x, 2)) = x

  sqrt_square3 : LEMMA sqrt(u * u) = abs(u)

  sqrt_square4 : LEMMA sqrt(expt(u, 2)) = abs(u)


  posreal_sqrt_is_positive: JUDGEMENT sqrt(px) HAS_TYPE posreal
```

```
--------------
  Properties
--------------

sqrt_zero : LEMMA  sqrt(0) = 0

sqrt_one : LEMMA  sqrt(1) = 1

null_sqrt : LEMMA  sqrt(x) = 0 IFF x = 0

sqrt_mult : LEMMA  sqrt(x * y) = sqrt(x) * sqrt(y)

sqrt_inv : LEMMA sqrt(1/px) = 1 / sqrt(px)

sqrt_div : LEMMA sqrt(x/py) = sqrt(x) / sqrt(py)



----------------
  Monotonicity
----------------

both_sides_sqrt : LEMMA sqrt(x) = sqrt(y) IFF x = y

both_sides_sqrt_lt : LEMMA sqrt(x) < sqrt(y) IFF x < y

both_sides_sqrt_le : LEMMA sqrt(x) <= sqrt(y) IFF x <= y

both_sides_sqrt_gt : LEMMA sqrt(x) > sqrt(y) IFF x > y

both_sides_sqrt_ge : LEMMA sqrt(x) >= sqrt(y) IFF x >= y




----------
  Bounds
----------

sqrt_lt1_bound : LEMMA  sqrt(x) < 1 IFF x < 1

sqrt_gt1_bound : LEMMA  sqrt(x) > 1 IFF x > 1
```

```
  sqrt_le1_bound : LEMMA  sqrt(x) <= 1 IFF x <= 1

  sqrt_ge1_bound : LEMMA  sqrt(x) >= 1 IFF x >= 1


  END square_root
```

The output will be as follows:

```
square_root : THEORY
BEGIN
        IMPORTING roots;
        x,y : VAR nonneg_real;
        px,py : VAR posreal;
        u,v : VAR real;
        sqrt( x ) : nonneg_real = root(x,2);
        sqrt_def : LEMMA sqrt(x) = y IFF y * y = x;
        square_sqrt : LEMMA sqrt(x) * sqrt(x) = x;
        sqrt_square : LEMMA sqrt(x * x) = x;
        square_sqrt2 : LEMMA expt(sqrt(x),2) = x;
        sqrt_square2 : LEMMA sqrt(expt(x,2)) = x;
        sqrt_square3 : LEMMA sqrt(u * u) = abs(u);
        sqrt_square4 : LEMMA sqrt(expt(u,2)) = abs(u);
        posreal_sqrt_is_positive : JUDGEMENT sqrt( px ) HAS_TYPE posreal;
        sqrt_zero : LEMMA sqrt(0) = 0;
        sqrt_one : LEMMA sqrt(1) = 1;
        null_sqrt : LEMMA sqrt(x) = 0 IFF x = 0;
        sqrt_mult : LEMMA sqrt(x * y) = sqrt(x) * sqrt(y);
        sqrt_inv : LEMMA sqrt(1 / px) = 1 / sqrt(px);
        sqrt_div : LEMMA sqrt(x / py) = sqrt(x) / sqrt(py);
        both_sides_sqrt : LEMMA sqrt(x) = sqrt(y) IFF x = y;
        both_sides_sqrt_lt : LEMMA sqrt(x) < sqrt(y) IFF x < y;
        both_sides_sqrt_le : LEMMA sqrt(x) <= sqrt(y) IFF x <= y;
        both_sides_sqrt_gt : LEMMA sqrt(x) > sqrt(y) IFF x > y;
        both_sides_sqrt_ge : LEMMA sqrt(x) >= sqrt(y) IFF x >= y;
        sqrt_lt1_bound : LEMMA sqrt(x) < 1 IFF x < 1;
        sqrt_gt1_bound : LEMMA sqrt(x) > 1 IFF x > 1;
        sqrt_le1_bound : LEMMA sqrt(x) <= 1 IFF x <= 1;
        sqrt_ge1_bound : LEMMA sqrt(x) >= 1 IFF x >= 1;
END square_root
```

1.6 Test3: Parity

```
 parity.pvs
 ----------------------------------------
  A few lemmas about odd or even numbers
 ----------------------------------------

parity : THEORY
  BEGIN
  i : VAR int

  incr_odd : LEMMA  odd?(i + 1) IFF even?(i)
  incr_even : LEMMA  even?(i + 1) IFF odd?(i)

  opposite_odd : LEMMA  odd?(-i) IFF odd?(i)
  opposite_even : LEMMA  even?(-i) IFF even?(i)

  parity1 : LEMMA odd?(i) OR odd?(i + 1)
  parity2 : LEMMA even?(i) OR even?(i + 1)

  odd_not_even : LEMMA  odd?(i) IFF NOT even?(i)
  even_not_odd : LEMMA  even?(i) IFF NOT odd?(i)

  even_2i : LEMMA  even?(2 * i)
  odd_2i_plus1 : LEMMA  odd?(2 * i + 1)
  odd_2i_minus1 : LEMMA odd?(2 * i - 1)

  -------------------------------------
    Rewrite rules for small constants
  -------------------------------------
  parity_zero: LEMMA even?(0)
  parity_one: LEMMA odd?(1)
  parity_two: LEMMA even?(2)
  parity_three: LEMMA odd?(3)
  parity_minus_one: LEMMA odd?(-1)
  parity_minus_two: LEMMA even?(-2)
  parity_minus_three: LEMMA odd?(-3)
END parity
```

The output after parsing will be:

```
parity : THEORY
BEGIN
```

```
        i : VAR int;
        incr_odd : LEMMA odd?(i + 1) IFF even?(i);
        incr_even : LEMMA even?(i + 1) IFF odd?(i);
        opposite_odd : LEMMA odd?(-i) IFF odd?(i);
        opposite_even : LEMMA even?(-i) IFF even?(i);
        parity1 : LEMMA odd?(i) OR odd?(i + 1);
        parity2 : LEMMA even?(i) OR even?(i + 1);
        odd_not_even : LEMMA odd?(i) IFF NOTeven?(i);
        even_not_odd : LEMMA even?(i) IFF NOTodd?(i);
        even_2i : LEMMA even?(2 * i);
        odd_2i_plus1 : LEMMA odd?(2 * i + 1);
        odd_2i_minus1 : LEMMA odd?(2 * i - 1);
        parity_zero : LEMMA even?(0);
        parity_one : LEMMA odd?(1);
        parity_two : LEMMA even?(2);
        parity_three : LEMMA odd?(3);
        parity_minus_one : LEMMA odd?(-1);
        parity_minus_two : LEMMA even?(-2);
        parity_minus_three : LEMMA odd?(-3);
END parity
```

Furthermore the prelude.pvs is used to test the parser as it contains all aspects of PVS.

2 Test for GUI

This part is the description of test for graphical user interface generated by JBuilder. Test cases for the user interface will be PVS theories and the test target will describe separately for each part of the user interface and at the end of the each part the figures will show the output result for each test target.

Test Requirements For GUI

- The theory should appear in the text area of the user interface. Theory designer can either select one theory from the file or type desire theory in the text editor area.
- Selected theory can be saved in a file. Any changes in the theory can also be saved in a file in the File|Save and File|Save as, menus.
- The font of the selected theory can be edited in the text area. Background color of the selected theory can also be edited in the edit menu.
- The text editor in the user interface have syntax highlighting. All PVS reserved words, symbols and identifiers have different colors.
- The tool bars buttons in the text area should be active: open file, clean the text area, save file and a link to PVS home page. The status reporting should also be active.

- The right click menu should be active in the text editor area. It can cut, copy, past, clear all, select all, edit font and background color of the selected theory in the text area.

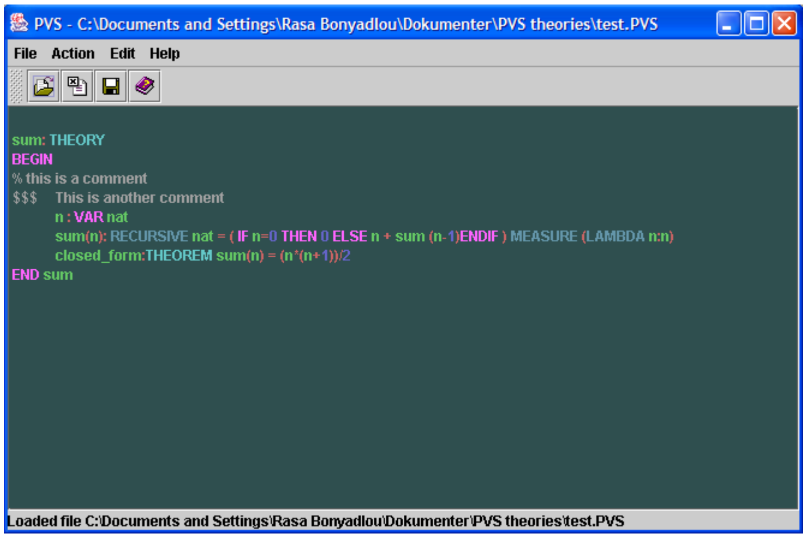
Test Data

Selected test data for the graphical user interface are PVS theories. For example theory designer can select "sum" theory as a test cases. Test cases:

```
sum: THEORY
  BEGIN
    n: VAR nat
    sum(n): RECURSIVE nat=
    (IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF)
    MEASURE (LAMBDA n:n)
    closed_form:THEOREM sum(n)= (n* (n+1)) / 2
  END sum
```

Test Instructions

- The "sum" theory has selected as a test cases for the PVS text editor. The selected theory appears in the text area. Figure 7.1 is the output for text editor. The selected theory is appear in the text area. The syntax highlighting is also visible in the text area.


PVS - C:\Documents and Settings\Rasa Bonyadlou\Dokumenter\PVS theories\test.PVS
File
Action
Edit
Help

```
sum: THEORY
BEGIN
% this is a comment
$$$     This is another comment
        n : VAR nat
        sum(n): RECURSIVE nat = ( IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF ) MEASURE (LAMBDA n:n)
        closed_form:THEOREM sum(n) = (n*(n+1))/2
END sum
```

Loaded file C:\Documents and Settings\Rasa Bonyadlou\Dokumenter\PVS theories\test.PVS

Figure 7.1: TextEditor

- Following is the procedure for different parts of the text editor. We got the test result by applying the test cases, "sum" theory, to the text editor.

Action	Results
Open File	Pass
Save File	Pass
Edit Font	Pass
Edit Color	Pass
Clean The Text Area	Pass
Tool Bar Buttons	Pass
Right Click Menu	Pass

3 Type checker

This section will describe the test instructions and requirements for the PVS type checker. Generally the PVS typecheck, detects the semantic errors in the PVS theories. Furthermore it checks whether the use of the names (such as variables, functions, type names) is inconsistent with their definition in the program. The test cases for type checking part are PVS theories. Following are the test instructions and results for PVS type checking.

3.1 Test Requirements

During type checking of PVS theories, if there is any semantic errors in the theories, an error message will be thrown. Following are some examples of the semantic errors that may occur in the PVS theories. The type checker will detect errors and will show an error message if any of following errors occur.

1. Semantic errors:

   - Wrong definition of types in theories.
   - Non-declared variables
   - Use of variable which is wrongly declared.
   - Duplicate declaration of variables/types/constants.

   Following are two other requirements for PVS type checker:

2. Compiled PVS files should be save with ".cpt" extension. The "cpt" files contain all the theories and symbol table of the compiled PVS file.

3. As described in the type checking design (chapter 6), PVS files may contain more than one theory, as a result of saving "cpt" files, all the theories in a pvs file will be saved in a single cpt file with the same name.

3.2 Test Instructions

This section will cover the test instruction and test cases for the test requirements. The first part will contain the theories which have semantic errors, type checker should detect the errors and throw an error message. The last part is a test case and output for "cpt" files.

Test Cases with Semantic Errors

Following are PVS theories that have the above semantic errors:

1. Non-declared variables "n".

```
typeCheckTest1: THEORY
BEGIN
   sum(n): RECURSIVE nat = ( IF n=0 THEN 0 ELSE n + sum (n-1)ENDIF ) MEASURE (LAMBDA n:n)
END typeCheckTest1
```

2. Test for duplicate declaration of variables.

```
typeCheckTest2: THEORY
 BEGIN
   n : VAR nat
   n : VAR nat
 END typeCheckTest1
```

3. Test for duplicate declaration of constants.

```
typeCheckTest3: THEORY
 BEGIN
    c : bool
    c : bool
    d : TYPE FROM int
    d : TYPE FROM int

 END typeCheckTest3
```

The type checker will throw an error message in all above theories.

Test Cases for "cpt" Files

Compiled PVS files should be save with ".cpt" extension. The "cpt" files contain all the theories and symbol table of the compiled PVS file. Following are test cases and output for the "cpt" files:

1. Test Cases: Extra-props

```
extra_props : THEORY
BEGIN
 IMPORTING absolute_value
 x1, x2, y1, y2 : VAR real
 e, e1, e2, d: VAR posreal
 n : VAR nat
 prod_bound : LEMMA
 abs(x1 - y1) < e1 AND abs(x2 - y2) < e2 IMPLIES
 abs(x1 * x2 - y1 * y2) < e1 * e2 + abs(y1) * e2 + abs(y2) * e1

 prod_epsilon : LEMMA
    EXISTS e1, e2 : e1 * e2 + abs(y1) * e2 + abs(y2) * e1 < e
 expt_continuous : LEMMA
     FORALL x1, e, n : EXISTS d : FORALL y1 :
        abs(y1 - x1) < d IMPLIES abs(expt(y1, n) - expt(x1, n)) < e

END extra_props
```

The generated "cpt" file is as follows:

```
//-------------------------------------------------//
//                 COMPILED PVS FILE               //
//-------------------------------------------------//
THEORY "extra_props"
TYPECHECK "false"
SYMBOLS 12
//-------------------------------------------------//
4 "x1"
4 "x2"
4 "y1"
4 "y2"
4 "e"
4 "e1"
4 "e2"
4 "d"
4 "n"
5 "prod_bound"
5 "prod_epsilon"
5 "expt_continuous"
//---------------------------------------------------//
```

2. Test cases: Parity

```
parity : THEORY
  BEGIN
  i : VAR int

  incr_odd : LEMMA  odd?(i + 1) IFF even?(i)
  incr_even : LEMMA  even?(i + 1) IFF odd?(i)
  opposite_odd : LEMMA  odd?(-i) IFF odd?(i)
  opposite_even : LEMMA  even?(-i) IFF even?(i)

  parity1 : LEMMA odd?(i) OR odd?(i + 1)
  parity2 : LEMMA even?(i) OR even?(i + 1)

  odd_not_even : LEMMA  odd?(i) IFF NOT even?(i)
  even_not_odd : LEMMA  even?(i) IFF NOT odd?(i)

  even_2i : LEMMA  even?(2 * i)
  odd_2i_plus1 : LEMMA  odd?(2 * i + 1)
  odd_2i_minus1 : LEMMA odd?(2 * i - 1)

  parity_zero: LEMMA even?(0)
  parity_one: LEMMA odd?(1)
  parity_two: LEMMA even?(2)
  parity_three: LEMMA odd?(3)
  parity_minus_one: LEMMA odd?(-1)
  parity_minus_two: LEMMA even?(-2)
  parity_minus_three: LEMMA odd?(-3)
END parity
```

The generated "cpt" file for this PVS theory is:

```
//-----------------------------------------//
//           COMPILED PVS FILE             //
//-----------------------------------------//
THEORY "parity"
TYPECHECK "false"
SYMBOLS 19
//-----------------------------------------//
4 "i"
5 "incr_odd"
5 "incr_even"
5 "opposite_odd"
5 "opposite_even"
5 "parity1"
5 "parity2"
5 "odd_not_even"
5 "even_not_odd"
5 "even_2i"
5 "odd_2i_plus1"
5 "odd_2i_minus1"
5 "parity_zero"
```

```
5 "parity_one"
5 "parity_two"
5 "parity_three"
5 "parity_minus_one"
5 "parity_minus_two"
5 "parity_minus_three"
//----------------------------------------//
```

Another Test Case for CPT

The PVS "analysis" file contains three separate theories, all the theories in the "pvs" file are saved in a single "cpt" file with the same name. Following is the PVS "analysis" file:

```
everything : THEORY
 BEGIN
  IMPORTING top_derivative, top_sequences, top_limits, top_continuity
 END everything
inverse_continuous_functions [ T1, T2 : NONEMPTY_TYPE FROM real ] : THEORY

BEGIN
  ASSUMING
  connected_domain : ASSUMPTION
     FORALL (x, y : T1), (z : real) :
  x <= z AND z <= y IMPLIES (EXISTS (u : T1) : z = u)
ENDASSUMING

 IMPORTING continuous_functions_props
  g : VAR { f : [T1 -> T2] | continuous(f) }
  inverse_incr : LEMMA
  bijective?[T1, T2](g) AND strict_increasing(g) IMPLIES
    continuous(inverse(g))

  inverse_decr : LEMMA
  bijective?[T1, T2](g) AND strict_decreasing(g) IMPLIES
    continuous(inverse(g))

  inverse_continuous : PROPOSITION
  bijective?[T1, T2](g) IMPLIES continuous(inverse(g))

END inverse_continuous_functions
top_derivative : THEORY
```

The output "cpt" file for the "analysis" theory will be as follows:

```
/--------------------------------------------------//
/               COMPILED PVS FILE                  //
/--------------------------------------------------//
THEORY "everything"
TYPECHECK "false"
SYMBOLS 0
/--------------------------------------------------//
/--------------------------------------------------//
THEORY "inverse_continuous_functions"
TYPECHECK "false"
SYMBOLS 6
/--------------------------------------------------//
2 "T1" 1
2 "T2" 1
4 "g"
5 "inverse_incr"
5 "inverse_decr"
5 "inverse_continuous"
/--------------------------------------------------//
THEORY "top_derivative"
TYPECHECK "false"
SYMBOLS 0
/--------------------------------------------------//
/--------------------------------------------------//
```

Overview

This chapter covered the test instructions, cases and results for PVS parser, GUI and type checker parts. All the requirements in these parts has been tested. Test cases are selected from PVS theories.

Chapter 8

Conclusion and future work

The aim of this project was, re-implementation of the *Prototyping Verification System* (PVS). This work has been done in two semester as a complete Master thesis. The aim of the first part of the project, was to take the initial steps towards the re-implementation of the *Prototyping Verification System.* And the aim of the second part of this project was to complete and build the PVS system. I started the work by studying the original PVS system which is distributed by SRI [1]. The architecture of PVS and the initial steps to start an automated proof in PVS were discussed. As a preparation for building the theorem prover, the logic system and the strategies for proving theories in PVS were discussed. It also covered different sorts of logic, syntax and semantic of each logic. A graphical user interface for PVS system was developed and tested. The PVS language and type system were discussed. By applying the whole PVS grammar, [8], to JavaCC, a parser for the PVS system was developed and tested. The main parts for PVS type checkering was also developed and tested. The theorem prover is not implemented, but the strategies behind the proof system and proof trees are explained and the classes for prover has been developed. Following are some ideas on future work in order to complete a full version of PVS verification system and improve this work:

- Implementation of theorem prover.
- In the graphical user interface:
  - Add search item to the user interface.
  - Localizing the user interface, in order to use it with different languages.
  - Designing a user interface for showing the proof steps and proof tree.
- In our parse tree, figure 6.7, the `datatype` part is ignored, so this part should also be implement in order to have a complete PVS type checker.
- Introduce tactics to the PVS prover system: more about sequent schema and the languages of sequent schema for derived rules mechanism.
- More about the Order Binary Trees and their application to PVS.

While working with PVS system, I acquired some experiences and ideas about the way to prove theories with an interactive theorem prover. Moreover I also gained experiences with PVS languages. During this project work I also acquired more experiences in Java programming language, in order to develop the system.

If we consider the time for developing the existing parts of the project, then an optimistic estimate for the time complete a version of PVS will be around two months. This time is for completing the reminding parts of the project, which are PVS prover and parts of the PVS type checker.

Bibliography

[1] Ricky W. Butler. An element tutorial on formal specification and verification using pvs2. Technical report, June 1995.

[2] John Rushby Natarajan Shankar Judy Crow, Sam Owre and Mandayam Srivas. A tutorial introduction to pvs. Technical report, SRI International, Menlo Park, CA, June 1995.

[3] Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. Technical report.

[4] Theodore S. Norvell. The javacc fqa. Technical report, Memorial University of Newfoundland, February 2002.

[5] S. Owre and N. Shankar. The formal semantic of pvs. Technical report, SRI International, Menlo Park, CA, August 1997.

[6] S. Owre and N. Shankar. The pvs prelude library. Technical report, SRI International, Menlo Park, CA, March 2003.

[7] J. M. Rushby S. Owre, N. Shankar and D. W. J. Stringer-Calvert. Pvs prover guide. Technical report, SRI International, Menlo Park, CA, September 1998.

[8] J. M. Rushby S. Owre, N. Shankar and D. W. J. Stringer-Calvert. Pvs language reference. Technical report, SRI International, Menlo Park, CA, December 2001.

[9] J. M. Rushby S. Owre, N. Shankar and D. W. J. Stringer-Calvert. Pvs system guide. Technical report, SRI International, Menlo Park, CA, December 2001.

[10] N. Shankar S. Owre and J. M. Rushby. User guide for the pvs specification and verification system. Technical report, SRI International, Menlo Park, CA, March 1993.

Appendix A

The Grammar

The complete PVS grammar which are used for implementation is presented in this Appendix. Below is the some points that should be consider for presentation of the syntax:

- Names in *italics* indicate syntactic classes and meta variables ranging over syntactic classes.
- The reserved words of the languages are in the `UPPERCASE`.
- An optional part of A of a clause is enclosed in square brackets:[A].
- Alternatives in a syntax production are separated by a bar |; a list of alternatives that is embedded in the right-hand side of a syntax production is enclosed in brackets, as in

  ```
  ExportingName  ::= IdOp [: { TypeExpr  | Type | FORMULA}]
  ```

- Iteration of a clause `B` one or more times is indicated by enclosing it in brackets followed by a plus sign : $B+$; repetition zero or more times is indicated by an asterisk instead of the plus sign: $B*$.
- A double plus or double asterisk indicates a clause separator; for example, $B**$',' indicates zero or more repetitions of the clause B separated by commas.

Specification

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

Importings and Exportings

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

Assumings

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

Theory Part

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

Declarations

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

Binding	::=	*TypedId* | { (*TypedIds*) }
TypedIds	::=	*IdOps* [: *TypeExpr*] [| *Expr*]
TypedId	::=	*IdOp* [: *TypeExpr*] [| *Expr*]

Type Expressions

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

Expressions

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

Expressions (continued)

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

Names

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

Identifiers

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

Datatypes

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

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

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

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

Appendix B

CD ROM

The CD ROM contains:

- The source code for the PVS GUI, parser, compiler and type checker, which is generated in JBuilder.
- The PVS executable version.
- Several PVS theories.