RAPID PROTOTYPING IN PVS*

César A. Muñoz†

ABSTRACT

PVSio is a conservative extension to the PVS prelude library that provides basic input/output capabilities to the PVS ground evaluator. It supports rapid prototyping in PVS by enhancing the specification language with built-in constructs for string manipulation, floating point arithmetic, and input/output operations.

1 INTRODUCTION

PVS [6] is a verification system based on a typed classical higher-order logic enriched with predicate subtyping and dependent records [7]. The system is widely known by its expressive specification language and its powerful theorem prover. The ground evaluator, which was originally announced as an experimental feature in PVS Release 2.3 [10], is a less known feature of PVS that allows for the animation of functional specifications: it extracts efficient Common Lisp code for a large set of PVS constructs [8].

The ground evaluator is a fundamental mechanism for rapid prototyping in PVS. However, it does not provide typical features of programming languages such as input/output functionality or floating-point arithmetic. For instance, a ground expression like `sqrt(2)`, where `sqrt` is defined as in the NASA reals library [1], is not handled by the ground evaluator. This lack of capabilities greatly limits the usefulness of the evaluator.

Another experimental feature of PVS, even less known than the ground evaluator, is called *semantic attachments* [4]. Semantic attachments are user-defined Common Lisp functions that the ground evaluator calls during the evaluation of PVS expressions. For instance, given an appropriate semantic attachment, the expression `sqrt(2)` can be evaluated as the Common Lisp expression `(sqrt 2)`. Semantic attachments must be handled with care. Indeed, the ground evaluator does not enforce type correctness of semantic attachments nor does it check that semantic attachments respect the intended semantics of the corresponding PVS expressions. Hence, semantic attachments may break the ground evaluator. However, since neither the ground evaluator nor the semantic attachments are integrated into the logical framework of PVS, the soundness of the theorem prover is not compromised.

PVSio is a prelude library extension implemented on top of semantic attachments that relieves PVS users from all the burden and technical details of Common Lisp programming of semantic attachments. PVSio enhances the specification language with built-in constructs for string manipulation, floating-point arithmetic, and input/output operations. From a logical point of view, PVSio is a conservative extension to the PVS prelude library.

*This work was supported by the National Aeronautics and Space Administration under NASA Cooperative Agreement NCC-1-02043.

†Senior Staff Scientist, National Institute of Aerospace (NIA), 144 Research Drive, Hampton, VA 23666. Email: munoz@nianet.org, Web: http://research.nianet.org/~munoz.

2 GROUND EVALUATOR

The PVS ground evaluator consists of a PVS to Common Lisp translator, an interactive read-eval-print interface, and a proof rule. The translation itself and the fragment of PVS that is suitable for ground evaluation are described in [10]. Roughly speaking, the unexecutable constructs of PVS are: uninterpreted symbols, non-bounded quantifications, and higher-order relations. However, evaluations are non-strict, that is, unexecutable terms may still occur in executable expressions as long as they are not required in the overall evaluation. The ground evaluator assumes that Type Correctness Conditions (TCCs) associated with executable expressions have been properly discharged. Unsound evaluations may result from unproven TCCs. For rapid prototyping, it is a working conjecture that ground evaluations are sound with respect to valid TCCs.

The interactive interface of the PVS ground evaluator is invoked with the Emacs command `M-x pvs-ground-evaluator`. It consists of a read-eval-print loop where user inputs are prompted by `<GndEval>`. Control commands such as `quit`, `help`, `quiet`, etc., may be typed after the prompt[1]. PVS expressions are supplied to the evaluator enclosed in double quotes (`"`). This notation is unfortunate since it requires PVS strings to be escaped when they are provided to the interactive evaluator interface. For instance, the PVS string `"Hello World"` has to be written `\"Hello World\"` within the read-eval-print loop.

The ground evaluator is also available through the proof rule `EVAL` in the theorem prover. The proof command `(EVAL` e`)` translates the ground expression e into Common Lisp, evaluates it, and displays the result. In contrast to expressions in the interactive environment, e is not enclosed in double quotes and, consequently, PVS strings may be used without escaping them. From a logical point of view, `EVAL` behaves like `SKIP` when it terminates. Henceforth, for presentation purposes, we will always refer to the interactive interface of the PVS ground evaluator. However, both interfaces provide similar functionality.

Assume the following definition of `sqrt_newton` from the theory `sqrt_approx` of the PVS NASA Langley reals library: [1]

```
sqrt_newton(a:nonneg_real,n:nat) : RECURSIVE posreal =
   IF n = 0 THEN a+1
   ELSE (1/2) * (sqrt_newton(a,n-1) + (a / sqrt_newton(a,n-1)))
   ENDIF
   MEASURE (n+1)
```

It has been formally verified that `sqrt_newton` satisfies among others the following properties:

1. $\sqrt{a} <$ `sqrt_newton`(a, n), and
2. `sqrt_newton`$(a, n+1) <$ `sqrt_newton`(a, n).

It can also be shown that when n goes to infinity, `sqrt_newton`(a, n) converges to $\sqrt{a}$. The function `sqrt_newton` can be animated in PVS:[2]

[1] See [10] for a description of the commands available at the interactive interface of the PVS ground evaluator.

[2] For readability, some messages have been supressed from the output.

```
<GndEval> "sqrt_newton(2,1)"
==>
11/6

<GndEval> "sqrt_newton(2,2)"
==>
193/132

<GndEval> "sqrt_newton(2,3)"
==>
72097/50952
```

We manually check that $\sqrt{2} < 72097/50952 < 193/132 < 11/6$. We soon realize that we cannot go too far with the evaluations:

```
<GndEval> "sqrt_newton(2,8)"
==>
60454027761135603009657451053482647417461350560672507323681148007853 2
81779195363708129280555186348570967518929738108196517997167224308353 8
46354049454543870962407321 7/
42747452979938782367552558491372955152442728254277762493955393004801 7
83198975672488286908591742962467078818766203444825272916779912617027 0
23054430820304915594482508 8
```

3 OVERVIEW TO PVSIO

For rapid prototyping we clearly need a better mechanism for exchanging information between the evaluation environment and PVS. At a minimum, one such mechanism should allow for basic input/output capabilities such as reading and pretty printing. Semantic attachments [4] provide the low level machinery needed for implementing this functionality. However, they are difficult to use, require Common Lisp knowledge and expertise, and are error prune.

PVSio is a PVS prelude library extension built on top of semantic attachments. It brings, among other things, basic input/output capabilities to the PVS ground evaluator without all the burden of semantic attachments and Common Lisp coding. For instance, after loading the PVSio prelude library[3], we can evaluate

```
<GndEval> "println(sqrt_newton(2,8))"
1.4142135
==>
TRUE
```

If we assume the following PVS definition:

[3]This is done with the Emacs command `M-x load-prelude-library PVSio`.

```
sqrt_io : void =
  LET a = query_real("Enter a positive real number:") IN
    assert(a >= 0,"Input Error") &
    println("The approx. sqrt of "+a+" is: "+sqrt_newton(a,8))
```

we can also evaluate

```
<GndEval> "sqrt_io"
Enter a positive real number:
2
The approx. sqrt of 2 is: 1.4142135
==>
TRUE

<GndEval> "sqrt_io"
Enter a positive real number:
1.5
The approx. sqrt of 1.5 is: 1.2247449
==>
TRUE

<GndEval> "sqrt_io"
Enter a positive real number:
-10
Assert Failure: Input Error
==>
FALSE
```

Note that the values returned by `println` and `sqrt_io` are Boolean. This is because `println` and `sqrt_io` are defined of type `void`, which is an alias to `bool` in PVSio. Hence, sequential statements can be separated with the symbol `&`, which is a logical *and* in PVS.

The function `sqrt_io` uses functions `query_real` and `assert`. The former allows for animation of expressions containing non-bounded variables that become dynamically bound to user inputs. PVSio does not check for consistency of user-provided input data. To partially overcome this potential source of problems, PVSio provides `assert(`b`,`s`)`, where b is a Boolean expression and s is a string. If b evaluates to `FALSE`, the message `"Assertion Failure: `s`"` is printed. From a logical point of view, the whole expression is equivalent to b.

We emphasize that for the PVS theorem prover there is nothing special about PVSio functions. For instance, the proposition $\forall(s)$: `query_real(`s`) = query_real(`s`)` is trivially proved in PVS. However, for the PVS ground evaluator, the value of the expression `query_real(`s`) = query_real(`s`)`, for a particular value of s, depends on the user inputs:

```
<GndEval> "query_real(s) = query_real(s)"
s
1
```

```
s
2
==>
FALSE
```

The example above shows that, as in the case of semantic attachments, PVSio functions may have side-effects in the evaluation environment. Hence, we may get unexpected results. However, these effects do no permeate to the PVS logical framework. In other words, `query_real(`s`) = query_real(`s`)` may evaluate to `FALSE` in the ground evaluator, but this cannot be used to prove `FALSE` in the theorem prover.

Input/output functionality is not the only limitation of the ground evaluator. Assume that we want to evaluate the ground expression $\sin(\sqrt{2})$. Similar to `sqrt_newton`, we may define an executable function `sin_approx` that converges to sine at infinity, for example using Taylor's series. In this case, we could animate `sin_approx(sqrt_newton(2))` to get a good approximation of $\sin(\sqrt{2})$. Such approximations of real and trigonometric functions are already defined in theories `sqrt_approx` and `trig_approx` of the NASA reals and trig libraries [1]. Functions in those theories are used in [2] to automate reasoning about non-computable functions such as squared root and trigonometric functions. However, they are not really suitable for rapid prototyping. Even the most efficient PVS definitions of `sqrt_newton` and `sin_approx` are not as efficient as native Common Lisp functions `sqrt` and `sin`. For this reason, PVSio enhances the ground evaluator with floating point arithmetic via semantic attachments:

```
<GndEval> "print(SIN(SQRT(2)))"
0.98776597
==>
TRUE
```

Floating point operators are fully capitalized in PVSio to emphasize that they are not the real functions. It is well-known that floating-point arithmetic may contradict standard mathematical results:

```
<GndEval> "SQ(SIN(2))+SQ(COS(2)) = 1"
==>
FALSE
```

4 PVSIO LIBRARY

PVSio[4] is implemented as a prelude library extension that profit from the library support provided in PVS 3.1 [11]. It consists of the following theories:

- `stdlang`: Language definitions.
- `stdstr`: String operations.
- `stdio`: Input/output operations.

[4]The PVSio package is freely available at http://research.nianet.org/~munoz/PVSio.

Table 1: *Theory* `stdlang`

Name	Type	Value	Description
void	`TYPE`	`bool`	Type of procedures and statements
skip	`void`	`TRUE`	Empty statement
exit	`void`	`FALSE`	Exit statement
assert`(b:bool,mssg:string)`	`void`	If `NOT b` then displays `mssg & exit`. Otherwise, `skip`	
assert`(b:bool,s:void)`	`void`	If `NOT b` then `s & exit`. Otherwise, `skip`	
assert`(b:bool)`	`void`	`assert(b,skip)`	
fail`(s:void)`	`void`	`s & exit`	
try`(s1:void,s2:void)`	`void`	`s1 OR s2`	
catch`(s:void)`	`void`	`s1 OR skip`	
seq`[T:TYPE](s:void,t:T)`	`T`	Performs `s`, returns `t`	

- `stdmath`: Floating point arithmetic.
- `stdpvs`: Basic reflection.
- `stdindent`: Definition of indentations.
- `stdtokenizer`: Definition of tokenizers.

Since PVSio is a prelude extension, the user does not need to import any of these theories. In a fresh context, the library PVSio is loaded with the Emacs command `M-x load-prelude-library PVSio`.

The rest of this section describes PVSio theories in some detail. However, note that PVSio is still under development. New theories and functions are expected to be added in the future.

4.1 Theory `stdlang`: Language definitions

Table 1 summarizes the definitions in theory `stdlang`.[5] Type `void` intends to serve as the type of procedures and statements, as opposed to functions and expressions. Constants `skip` and `exit`, both of them of type `void`, represent the empty and the exit statements, respectively. Since `void` is equivalent to `bool`, the symbol `&` (logical *and*) serves as separator of sequential statements. Furthermore, quantifiers `FORALL` and `EXISTS`, over a finite domain, serve as iterators. PVSio does not provide assignments as statements.

Theory `stdlang` also defines control structures such as `assert`, `try`, `fail`, `catch`, and `seq`. As we saw in Section 3, `assert` provides a mechanism for simple runtime verification of Boolean conditions. Operations `fail`, `try`, and `catch` complement the functionality of `assert` with basic handling of exceptional cases. Finally, `seq` allows for sequencing of statements, possibly having side-effects, and PVS expressions. Type `T` is usually inferred by PVS, but in some cases it has to be provided by the user.

[5] For technical reasons, `assert(b:bool,mssg:string)` is defined in theory `stdio`, and `seq` is defined in theory `stdproc`.

Table 2: *Theory* **stdstr**

Name	Type	Description
empty	string	Empty string
space	string	Space
newline	string	New line
tab	string	Tab
quote	string	Double quote
spaces(n:nat)	string	n spaces
upcase(s:string)	string	Uppercase of s
downcase(s:string)	string	Lowercase of s
capitalize(s:string)	string	Capitalization of s
substr(s:string,i,j:nat)	string	Substring of s from i to j-1
find(s1,s2:string)	int	Index of leftmost occurrence of s2 in s1 (or -1)
fill(n,s:string)	string	n times string s
str2real(s:string)	rat	Rational denoted by s (s is in decimal notation)
str2int(s:string)	int	Integer denoted by s
str2bool(s1,s2:string)	bool	downcase(s1) = downcase(s2)
number?(s:string)	bool	Tests if string s represents a numerical value
int?(s:string)	bool	Tests if string s represents an integer value
concat(s1,s2:string)	bool	Concatenation of s1 and s2
#(s:string)	rat	str2real(s)
tostr(e)	string	String representation of e
e_1 + e_2	string	concat(e_1,e_2)

Note that most functions in `stdlang` are limited to statements, i.e., expressions of type `void`. PVSio does not waive PVS type checking. For example, if the domain of a function is `real`, it cannot return `exit`. Furthermore, `stdlang` operations do not extend to Common Lisp exceptions. Currently, PVSio does not provide a mechanism for handling Common Lisp exceptions.

4.2 Theory `stdstr`: String operations

Table 2 summarizes the definitions in theory `stdstr`. Note that PVS does not support decimal notation of real numbers. PVSio alleviates this limitation with the function `str2real` (also denoted with the symbol `#`). It takes the string representation of a number in decimal notation and returns its value as a rational number. For instance, the string `#("1.5")` evaluates to the PVS number `3/2`.

The symbols `tostr` and `+` are overloaded for basic values, that is, e, e_1, e_2 may be of type `real`, `bool`, or `string`. Since `tostr` is a PVS conversion, basic values are automatically coerced to string values when necessary. Indeed, the symbol `+` behaves pretty much as `+` in Java. The practice of having conversions from user-defined types to `string` is strongly advised to fully exploit the functionality provided by `stdstr`.

Examples:

```
<GndEval> "#(\"1.5\") + #(\"2.3\") - 1"
==>
14/5

<GndEval> "1+tab+12+tab+123+tab+1234+tab+12345+tab"
==>
"1       12      123     1234    12345   "

<GndEval> "\"Boolean: \"+(1=1)"
==>
"Boolean: TRUE"
```

4.3 Theory `stdio`: Input/Output operations

Table 3 summarizes the definitions in theory `stdio`. This theory implements classical I/O functionality as reading from standard input, printing to standard output, and reading and writing to character streams. As said before, PVSio does not check for consistency of input data. Hence, the user is responsible for providing the appropriate kind of data, otherwise the ground evaluator will break.

Input operations are based on Common Lisp reading conventions, which are fairly complex (see [9], Section Input/Output). For the sake of simplicity, we assume the following lexical rules:

- Integer: Sequence of numerical characters.
- Real: Sequence of numerical characters and dot symbol (`.`).
- Word: Sequence of alpha-numerical characters and symbols in `!?@#$%^&_*/+-=|{}[]<>` separated by `space`, `newline`, or `tab`.
- Line: Sequence of symbols terminated by `newline`.

Symbols in `(),.:;\~‘’"` have special meaning in Common Lisp and they are not handled correctly by PVSio.

Character streams are defined by an uninterpreted type `Stream`. Input and output streams, `IStream` and `OStream`, respectively, are defined as subtypes of `Stream`. A stream is typically associated with a file. In this case they are called *file streams*. PVSio also allows *string streams*, that is, streams associated with strings. Except for the functions `flength`, that only applies to file streams, and the function `sget`, that only applies to output string streams, both kinds of streams are handled indistinctly by PVSio.

An input stream can be created in two modes: `input` (which is the default) and `string`. For instance, `open_in(input,`s`)` opens an input file stream associated to the file named s. PVSio assumes that the file does exist. The existence of such a file can be tested with `fexists(`s`)`. The operation `open_in(string,`s`)` opens an input string stream that reads information from string s.

Table 3: *Theory* `stdio`

Name	Type	Description
Standard Input		
query_int(s:string)	int	Prompts s, reads and returns an integer
query_real(s:string)	rat	Prompts s, reads a decimal and returns it as rat
query_word(s:string)	string	Prompts s, reads and returns a string word
query_line(s:string)	string	Prompts s, reads and returns a string line
query_bool(s1,s2:string)	bool	str2bool(query_word(s1),s2)
read_int	int	query_int(empty)
read_real	rat	query_real(empty)
read_word	string	query_word(empty)
read_line	string	query_line(empty)
read_bool(s:string)	bool	query_bool(empty,s)
Standard Output		
print(s:string)	void	Prints string s
println(s:string)	void	Prints string s+newline
I/O Character Streams		
Stream	TYPE+	Uninterpreted type of character streams
IStream	$\subseteq$ Stream	Uninterpreted type of input character streams
OStream	$\subseteq$ Stream	Uninterpreted type of output character streams
Mode	TYPE	{input,new,append,overwrite,string}
stdin	IStream	Standard input stream
stdout	OStream	Standard output stream
open_in(m:Mode,s:string)	IStream	Opens an input stream
open_in(s:string)	IStream	open_in(input,s)
open_out(m:Mode,s:string)	OStream	Opens an output stream
open_out(s:string)	OStream	open_out(new,s)
sget(f:OStream)	string	String from output string stream f
close(f:Stream)	void	Closes f
eof?(f:Stream)	bool	Tests the end of stream f
flength(f:Stream)	nat	Length of file stream f
fexists(s:string)	bool	Tests if there exists a file with name s
Input from Character Streams		
scan_int(f:IStream)	int	Reads an integer from f and returns it
scan_real(f:IStream)	rat	Reads a decimal from f and returns it as rat
scan_word(f:IStream)	string	Reads a string word from f and returns it
scan_line(f:IStream)	string	Reads a string line from f and returns it
scan_bool(f:IStream,s:string)	bool	str2bool(scan_word(f),answer)
Output to Character Streams		
print(f:OStream,s:string)	void	Writes string s to f
println(f:OStream,s:string)	void	Writes string s+newline to f
echo(f:OStream,s:string)	void	Writes string s to f and to stdout
echoln(f:OStream,s:string)	void	Writes string s+newline to f and to stdout

An output stream can be created in several modes: `new` (which is the default), `append`, `overwrite`, and `string`. All the modes in {`new`,`append`,`overwrite`} open output file streams. They differ in the action they perform should a file with the same name already exist. The mode `new` supersedes any file with the same name, the mode `append` writes at the end of the old file, and the mode `overwrite` writes at the beginning of the old file overriding the content of the file as output operations are performed. The operation `open_out(string,`s`)` opens an output string stream (the value of s is ignored). If f is an output string stream, the content of the string that is being written is obtained with `sget(`f`)`.

Examples:

```
<GndEval> "read_line"
Hello World
==>
"Hello World"

<GndEval> "read_int + read_real"
10
10.5
==>
41/2

<GndEval> "LET i=query_int(\"Give me a nat:\") IN print(1+spaces(i)+1)"
Give me a nat:
4
1    1
==>
TRUE

<GndEval> "println(\"Hello World\")"
Hello World
==>
TRUE

<GndEval> "print(1=0) & println(123) & println(1/2)"
FALSE123
0.500000
==>
TRUE
```

4.4 Theory `stdmath`: Floating point arithmetic

Table 4 summarizes the definitions in theory `stdmath`. Names are fully capitalized to emphasize that they are floating point operations (not the real ones). The constant `NaN` is returned by functions in `stdmath` when the result is not well-defined, e.g., `SQRT(-1) = NaN`. However, since `NaN` is not a ground expression, it cannot be evaluated by the PVS ground evaluator. `NaN` may still be part of an executable expression when its value is not required in the overall evaluation.

Table 4: *Theory* `stdmath`

Name	**Type**	**Description**
NaN	`number`	Uninterpreted number
SQ(`x:real`)	`nnreal`	$\mathtt{x}^2$
SQRT(`x:real`)	`nnreal`	$\approx \sqrt{\mathtt{x}}$
SIN(`x:real`)	`real`	$\approx \sin \mathtt{x}$
COS(`x:real`)	`real`	$\approx \cos \mathtt{x}$
TAN(`x:real`)	`real`	$\approx \tan \mathtt{x}$
ASIN(`x:real`)	`real`	$\approx \sin^{-1} \mathtt{x}$
ACOS(`x:real`)	`real`	$\approx \cos^{-1} \mathtt{x}$
ATAN(`y,x:real`)	`real`	$\approx \tan^{-1} \mathtt{x/y}$
PI	`real`	$\approx \pi$
NPI(`n:int`)	`real`	$\approx n\pi$
RANDOM	`real`	Pseudo-random number in the real interval $[0, 1)$
NRANDOM(`n:posnat`)	`nat`	Pseudo random number in the natural interval $[0, n)$
BRANDOM	`bool`	Pseudo random Boolean
RAD2DEG(`x:real`)	`real`	$\approx 180x/\pi$
DEG2RAD(`x:real`)	`real`	$\approx x\pi/180$
TRUNC(`x:real`)	`int`	Truncation of x
ROUND(`x:real`)	`int`	Rounding of x
DIV(`n:nat,m:posnat`)	`nat`	Euclidean division of `n` by `m`
MOD(`n:nat,m:posnat`)	`nat`	Remainder of Euclidean division of `n` by `m`

Table 5: *Theory* ***stdpvs***

Name	Type	Description
str2PVS`[T](s:string)`	`T`	Translates a string to a PVS expression
PVS2str`(type:string,t:T)`	`string`	Translates a PVS expression to a string

4.5 Theory `stdpvs`: Reflection

PVSio provides very basic capabilities for reflection through the theory `stdpvs` (Table 5). Strings are translated back and forth to PVS via `str2PVS` and `PVS2str`. For example, assuming the following definitions in PVS

```
REC : TYPE = [# x : real, y: real #]

rec : REC = (# x := 5, y:= 6 #)
```

we get

```
<GndEval> "str2PVS[REC](\"rec\")"
==>
(# x := 5, y := 6 #)

<GndEval> "PVS2str(\"REC\",rec)"
==>
"(# x := 5, y := 6 #)"
```

Note that the type of the expression is given as a theory parameter in `str2PVS`, whereas it is given as a string in `PVS2str`. Full reflection in PVSio is limited by the fact that the PVS ground evaluator ignores all type information from PVS expressions when translating them to Common Lisp. Therefore, there is no a simple way to reconstruct a PVS type from the Common Lisp translation of a PVS ground expression.

4.6 Theory `stdindent`: Indentations

Theory `stdindent` (Table 6) defines *indentations*. An indentation is composed of a built-in stack of integers, a predefined number of spaces, and a prefix. Indentations are suitable for printing block-formatted text, such as source code, or linearization of tree-structures. The use of indentation is best illustrated with some examples.[6]

Assume the definitions

```
chello : void =
   LET i = create_indent(2) IN
   prindentln(i,"void main() {") & open_block(i) &
   prindentln(i,"while(1) {") & open_block(i,3) &
   prindentln(i,"println("+quote+"Hello World"+quote+");") & close_block(i) &
```

[6] These examples are available as part of the standard distribution of PVSio.

Table 6: *Theory* ***stdindent***

Name	Type	Description
Indent	TYPE+	Uninterpreted type for indentations
create_indent(n:nat,s:string)	Indent	Indentation of n spaces and prefix s
create_indent(n:nat)	Indent	Indentation of n spaces without prefix
open_block(i:Indent,n:nat)	void	Opens block of n spaces
open_block(i:Indent)	void	Opens block with default spaces
close_block(i:Indent)	void	Closes block
get_indent(i:Indent)	nat	Default number of spaces
set_indent(i:Indent,n:nat)	void	Sets the default number of spaces to n
get_prefix(i:Indent)	string	Default prefix
set_prefix(i:Indent,s:string)	void	Sets the default prefix to s
indent(i:Indent,s:string)	string	Indented s
prindent(i:Indent,s:string)	void	Prints indented s
prindentln(i:Indent,s:string)	void	Prints indented s + newline
prindent(f:OStream,i:Indent,s:string)	void	Writes indented s to f
prindentln(f:OStream,i:Indent,s:string)	void	Writes indented s + newline to f

```
   prindentln(i,"}") & close_block(i) &
   prindentln(i,"}")

stair : void =
  LET i = create_indent(3,"=>") IN
  (FORALL(n:subrange(1,3)): prindentln(i,n) & open_block(i)) &
  set_indent(i,1) &
  (FORALL(n:subrange(4,6)): prindentln(i,n) & open_block(i)) &
  set_prefix(i,"<=") &
  (FORALL(n:subrange(0,6)): prindentln(i,n) & close_block(i))
```

Then,

```
<GndEval> "chello"
void main() {
  while(1) {
     println("Hello World");
  }
}
==>
TRUE

<GndEval> "stair"
=>1
=>   2
```

```
=>       3
=>          4
=>           5
=>             6
<=               0
<=             1
<=           2
<=          3
<=       4
<=    5
<=6
==>
TRUE
```

4.7 Theory `stdtokenizer`: Tokenizers

Theory `stdtokenizer` (Tables 7 and 8) is the most elaborate theory of `PVSio`. By itself, `stdtokenizer` is a good example of what can be achieved with the PVSio package. Indeed, the entire theory is written as a `PVSio` application.

A *tokenizer* is a list of *tokens*, i.e., string words, with basic parsing capabilities. The structure of the type `Tokenizer` is not relevant here. As structured programming mandates, users should access the internal structure only through the public interface. Tokenizers can be created from files through `file2tokenizer(`s`)`, where s is the name of the file, or from strings through `str2tokenizer(`s`)`, where s is the string containing the tokens. Once a tokenizer is created, `go_next` and several `accept`-functions process the tokens one by one. Errors are indicated by `error?(`t`)` and `error(`t`)`. User-defined errors are allowed through `set_error(`t`,`*code*`)`, where negative codes are reserved for this purpose. Standard error codes are listed in Table 9.

Error messages are available through `Messenger` objects, which are functions from `int` to `string`. Standard error messages for tokenizer t are retrieved with function `std_mssg(`t`)`. The statement `print_error(`t`)` prints an error message for t from the standard messenger. Furthermore, standard errors messages can be overridden by using *mssg* `* (`*code*`,`s`)`. If *mssg* is a messenger, `print_error(`t`,`*mssg*`)` prints an error message for t from *mssg*.

We illustrate the use of `stdtokenizer` with a simple example.[7] Assume that we want to parse a list of natural numbers given as a string `"<` n_1 `&` ... `&` n_k `>"`, where $1 \leq n_i \leq 5$ for $0 \leq i \leq k$ ($i = 0$ being the empty list). The PVS theory `parse` in Figure 1 implements that informal specification (lines have been numbered for the discussion).

- Line 4 defines the test that numbers in the list should satisfy, i.e., $1 \leq n_i \leq 5$.
- Lines 5 to 10 define error messages to be printed in case of a syntax error. We have decided to override standard messages for `EndOfTokenizer` and `ExpectingTestWord`, and to add new codes -1 and -2 for special error messages.
- The parser itself is implemented in lines 11 to 30 as a tail recursive function. The parameters of that function are the current tokanizer `t` and the accumulated list `l`.

[7] The example is available as part of the standard distribution of PVSio.

Table 7: *Theory* ***stdtokenizer****: Declarations*

Name	Type
Tokenizer	TYPE
file2tokenizer(s:string)	Tokenizer
str2tokenizer(s:string)	Tokenizer
tostr(t:Tokenizer)	string
error(t:Tokenizer)	int
error?(t:Tokenizer)	bool
set_error(t:Tokenizer,code:int)	Tokenizer
token(t:Tokenizer)	string
next_token(t:Tokenizer)	string
token_at(t:Tokenizer,n:nat)	string
val_int(t:Tokenizer)	int
val_real(t:Tokenizer)	real
eot?(t:Tokenizer)	bool
line(t:Tokenizer)	nat
length(t:Tokenizer)	nat
pos(t:Tokenizer)	upto(length(t))
go_next(t:Tokenizer)	Tokenizer
go_back(t:Tokenizer)	Tokenizer
accept_word(t:Tokenizer,test:[string->bool])	Tokenizer
accept_word(t:Tokenizer,s:string)	Tokenizer
accept_word(t:Tokenizer)	Tokenizer
accept_int(t:Tokenizer,test:[int->bool])	Tokenizer
accept_int(t:Tokenizer)	Tokenizer
accept_real(t:Tokenizer,test:[real->bool])	Tokenizer
accept_real(t:Tokenizer)	Tokenizer
Messenger	TYPE
std_mssg(t:Tokenizer)	Messenger
***(mssg:Messenger,(code:int,s:string))	Messenger
print_error(t:Tokenizer)	void
print_error(t:Tokenizer,mssg:Messenger)	void

Table 8: *Theory* ***stdtokenizer****: Descriptions*

Name	**Description**
file2tokenizer(*s*)	Creates a tokenizer from file named *s*
str2tokenizer(*s*)	Creates a tokenizer from string *s*
tostr(*t*)	String representation of tokenizer *t*
error(*t*)	Error code of the tokenizer
error?(*t*)	Tests if tokenizer has found a syntax error
set_error?(*t*, *code*)	Sets code error to *code*
token(*t*)	Current token (token already processed)
next_token(*t*)	Next token (next token to process)
token_at(*t*, *n*)	Token at position *n*
val_int(*t*)	Integer value of current token or 0
val_real(*t*)	Real value of current token or 0
eot?(*t*)	Tests for end of tokenizer
line(*t*)	Line number of current token
length(*t*)	Length of tokenizer
pos(*t*)	Current position of tokenizer
go_next(*t*)	Processes next token
go_back(*t*)	Goes to previous processed token
accept_word(*t*, *test*)	Accepts word that satisfies *test*
accept_word(*t*, *s*)	Accepts word *s*
accept_word(*t*)	Accepts any non-numerical value
accept_int(*t*, *test*)	Accepts integer that satisfies *test*
accept_int(*t*)	Accepts any integer
accept_real(*t*, *test*)	Accepts real that satisfies *test*
accept_real(*t*)	Accepts any real
Messenger	`[int->string]`
std_mssg(*t*)	Standard messenger for *t*
mssg * (*code*, *s*)	Overrides messenger *mssg* with message *s* in *code*
print_error(*t*)	Prints error message from `std_mssg`
print_error(*t*, *mssg*)	Prints error message from messenger *mssg*

```
 1: parse : THEORY
 2: BEGIN

 3:  IMPORTING list[int]

 4:  test15(n:int):bool = 1 <= n AND n <= 5

 5:  mssg15(t:Tokenizer):[int->string] =
 6:    std_mssg(t) *
 7:    (EndOfTokenizer,"Truncated list") *
 8:    (ExpectingTestWord,"Expecting <. Found: "+next_token(t)) *
 9:    (-1,"Expecting 1 <= n <= 5. Found: "+next_token(t)) *
10:    (-2,"Expecting either & or >. Found: "+next_token(t))


11:  parse15(t:Tokenizer,l:list) : RECURSIVE [Tokenizer,list] =
12:    IF error?(t) THEN (t,null)
13:    ELSIF eot?(t) THEN (set_error(t,EndOfTokenizer),null)
14:    ELSIF number?(next_token(t)) THEN
15:      LET t = accept_int(t,test15) IN
16:      IF error?(t) THEN
17:        (set_error(t,-1),null)
18:      ELSIF next_token(t) = "&" THEN
19:        parse15(accept_word(t,"&"),cons(val_int(t),l))
20:      ELSIF next_token(t) = ">" THEN
21:        (t,reverse(cons(val_int(t),l)))
22:      ELSE
23:        (set_error(t,-2),null)
24:      ENDIF
25:    ELSIF null?(l) AND next_token(t) = ">" THEN
26:      (t,null)
27:    ELSE
28:      (set_error(t,-1),null)
29:    ENDIF
30:    MEASURE IF error?(t) THEN 0 ELSE (length(t)-pos(t)+1) ENDIF

31:  parser(s:string) : list =
32:     LET t = str2tokenizer(s) IN
33:     LET (t,l) = parse15(accept_word(t,"<"),null) IN
34:     seq(print_error(t,mssg15(t)),l)

35: END parse
```

Figure 1: *Example of* ***stdtokenizer*** *capabilites*

Table 9: *Error codes for* ***stdtokenizer***

Code	Value	Description
NoError	0	No error has been found
FileNotFound	1	File not found (name of the file available at `next_token(`t`)`)
EndOfTokenizer	2	End of tokenizer has been found
InvalidToken	3	Invalid token
ExpectingWord	4	Expecting a word
ExpectingTestWord	5	Expecting word that satisfies test
ExpectingInt	6	Expecting an integer
ExpectingTestInt	7	Expecting integer that satisfies test
ExpectingReal	8	Expecting a real
ExpectingTestReal	9	Expecting real that satisfies test
	$n < 0$	Reserved for user-defined errors

After processing the tokens, the function returns a tokenizer and the constructed list (which has been reversed in line 21 to return it in the right order). Line 30 gives the measure that guarantees termination.

- Lines 31 to 34 define a simple function that tests the parser. It has as input the string to be processed and as output the list that has been constructed. If the parser finds a syntax error an error message is printed and the list `null` is returned. If instead of reading the tokens from a string, we would like to read them from a file called s, we simply write `LET t = file2tokenizer(`s`) IN` in line 32.

Finally, we test our parser with some examples:

```
<GndEval> "parser(\"< >\")"
==>
(: :)

<GndEval> "parser(\"< 5 >\")"
==>
(: 5 :)

<GndEval> "parser(\"< 2 & 4 & 3 & 1 >\")"
==>
(: 2, 4, 3, 1 :)

<GndEval> "parser(\"< 2 4 3 1 \")"
Syntax Error. Line 1: Expecting either & or >. Found: 4
==>
(: :)
```

```
<GndEval> "parser(\"< 20 >\")"
Syntax Error. Line 1: Expecting 1 <= n <= 5. Found: 20
==>
(: :)
```

The example above illustrates the functionality provided by theory `stdtokenizer`. However, the simplest way of parsing PVS expressions, such as lists and data types, is using the standard PVS lexer and parser via `str2PVS` (see Section 4.5). In that case, syntax and type-checking errors are reported directly by PVS, and, therefore, the user has no control on the error handling mechanism.

5 CONCLUSION

Rapid prototyping is the ability to execute a system from the very early stages of its development. Formal techniques address rapid prototyping by extracting code from formal specifications. The PVS ground evaluator, for instance, extracts very efficient Common Lisp code from PVS specifications. If we trust the implementation of the verification system (including the theorem prover and the ground evaluator), the PVS ground evaluator produces a correct and efficient Common Lisp implementation from the functional specification of an algorithm,

Extraction of efficient code is not enough for rapid prototyping. Usually, formal specifications languages, such as PVS, include a large subset of executable functions that may act as a functional programming language. However, they also miss features that are essential in programming languages, such as input/output operations, exception handling, floating point arithmetic, and side-effects. In general, these limitations are addressed in extraction tools by adding some degree of informality to the system, such as *realizations* in Coq [12], or *semantic attachments* in PVS [4]. Due to this informality, the extracted code may be inconsistent with respect to the original specification.

With these limitations in mind, we have developed PVSio to extend the PVS formal specification language with standard programming language features. Since PVSio is based on semantic attachments, extracted code may not respect its original specification. However, PVSio has been designed to minimize logical errors that may be introduced by semantic attachments. First of all, PVSio is a conservative extension to the PVS prelude library. There are no axioms in PVSio, therefore, `FALSE` cannot be proved with the help of PVSio theories. Yet, expressions like `RANDOM = RANDOM` may evaluate to `FALSE` in the PVS ground evaluator, while they are provable `TRUE` in the PVS theorem prover. However, since there is a clean separation between the theorem prover and the ground evaluator, `FALSE` cannot be proved from PVSio evaluations. Finally, PVSio has been designed to be minimally invasive. That is, sources of problems such as side-effects are, by design, well-localized in PVSio applications.

There are two logical weakness in PVSio applications: floating point arithmetic and input operations. It is well-known that floating point arithmetic does not satisfy even simple real arithmetic properties such as associativity. For that reason, functions in `stdmath` are not identified with functions in the `reals` library. In other words, PVSio does not have axioms such as `SQ(SIN(x))+SQ(SIN(x))=1`. On the negative side, there are no useful properties on functions defined in `stdmath`. We expect to integrate a PVS formalization of floating point

arithmetic [3] with PVSio to allow reasoning about programs that use `stdmath`.

Side-effects produced by output operations are ignored at the logical level. For example, `print(`s`)` is logically equivalent to `TRUE` in PVS. On the other hand, input operations are problematic because their evaluations are not statically determined. This problem is classically solved by structuring the programs such that there is a phase for input and a phase for processing data. The theory `stdtokenizer` is a good example of this method. Except for the initializations of tokenizers that use input operations, all the other operations are purely functional, i.e., they do not have side effects. Hence, we may formally reason about a function like `parse15` in PVS. Indeed, we have proved that `parse15` terminates for *any* tokenizer.

PVSio does not intend to be another programming language. It has been designed to serve as a safe and efficient rapid prototyping tool for PVS. It lacks functionalities such as general side-effects, i.e., assignments, low-level exception handling, and unbounded (and possibly non-terminating) loops. These constructs are too invasive to be incorporated in a functional specification language such as PVS. Functional counterparts of these constructs such as let-in expressions, defensive programming, and recursive functions seem more appropriate for this kind of specification language. Despite its limitations, PVSio is a powerful tool that may handle non-trivial examples, such as a Conflict Detection and Resolution algorithm for Air Traffic Management developed at the National Institute of Aerospace (formerly ICASE) and NASA Langley [5]. In the future, new theories and functions are expected to be added. We are particularly interested on supporting the interface of PVS with other environments through PVSio.

ACKNOWLEDGMENTS

The idea for PVSio was inspired by examples of semantic attachments from [4]. We are specially thankful to Jonh Rushby for giving us an early draft of that paper.

REFERENCES

[1] Formal Methods Groups at NASA Langley and National Institute of Aerospace. NASA langley PVS libraries. Available at http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS2-library/pvslib.html.

[2] R. Butler, V. Carreño, G. Dowek, and C. Muñoz. Formal verification of conflict detection algorithms. In *Proceedings of the 11th Working Conference on Correct Hardware Design and Verification Methods CHARME 2001*, volume 2144 of *LNCS*, pages 403–417, Livingston, Scotland, UK, 2001. A long version appears as report NASA/TM-2001-210864.

[3] V. Carreño and P. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *HOL95: Eighth International Workshop on Higher-Order Logic Theorem Proving and Its Applications*, Aspen Grove, UT, September 1995. Category B proceedings. Available at http://lal.cs.byu.edu/lal/hol95/Bprocs/indexB.html.

[4] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001. Available at http://www.csl.sri.com/users/rushby/abstracts/attachments.

[5] G. Dowek, C. Muñoz, and A. Geser. Tactical conflict detection and resolution in a 3-D airspace. Technical Report NASA/CR-2001-210853 ICASE Report No. 2001-7, ICASE-NASA Langley, ICASE Mail Stop 132C, NASA Langley Research Center, Hampton VA 23681-2199, USA, April 2001.

[6] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[7] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[8] N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1999. Available at http://www.csl.sri.com/shankar/PVSeval.ps.gz.

[9] G. L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, MA, second edition, 1990.

[10] PVS Development Team. PVS 2.3 experimental features – ground evaluation. Available at http://pvs.csl.sri.com/experimental/eval.html.

[11] PVS Development Team. PVS 3.1 release note. Available at http://pvs.csl.sri.com/pvs-release-notes.html.

[12] The Coq Team. The Coq proof assistant: Reference manual: Version 7.2. Technical Report RT-0255, INRIA, Rocquencourt, France, February 2002. Available at http://coq.inria.fr/doc/main.html.