Universidad Nacional de San Luis

República Argentina

Transforming RSL into PVS

Aristides Dasso

Asesor: **Chris George**

Mayo 2002

Tesis de Maestría en Ingeniería del Software

Universidad Nacional de San Luis

República Argentina

Ejećito de los Andes 950
5700 San Luis
Argentina

Transforming RSL into PVS

Aristides Dasso

Asesor: **Chris George**

Abstract

Transformation from one formal language to another is a challange, specially when they differ conceptually. In that case these differences become a challenge and we are obliged to solve theoretical problems before the transformation can be done. We describe in this report some of these problems and how they can be overcome, ilustrating the problematic issues using two formal languages: RSL and PVS. Both —PVS and RSL— are formal languages that can be used to do formal specifications of complex software systems. And altough both languages have some similarities they differ in aspects that are crucial —RSL is a much bigger language than PVS and moreover has several constructs that make the languages substantially, and in some cases, conceptually different (partial as well as total functions against only total ones, etc.). The description here includes solutions to some of these problems as well as a the details of the design of a tool that automatically translates RSL into PVS. There is an added benefit to this namely the fact that altough both methods have a prover tool, PVS's is free, so a translator from RSL to PVS would allow a specifiaction written in RSL to be proved in PVS using the freely available PVS prover.

Tesis de Maestría en Ingeniería del Software

Chris George is a Senior Research Fellow at UNU/IIST, 1 September 1994 - 31 August 2003. He is one of the main contributors to RAISE, particularly the RAISE method, and that remains his main research interest. Before coming to UNU/IIST he worked for companies in the UK and Denmark.

Contents

List of Tables

List of Figures

1 Introduction

Transformation from one formal language to another is a challenge, specially when they differ in aspects that are conceptually different. In that case these differences become an obstacle and we are obliged to solve theoretical problems before the transformation can be done.

We describe in this report some of these problems and how they can be overcome, illustrating the problematic issues using two formal languages: RSL and PVS.

Prototype Verification System (PVS) and RAISE are both examples of Formal Methods.

"PVS is a *Prototype Verification System* for the development and analysis of formal specifications. The PVS system consists of a specification language, a parser, a type checker, a prover, specification libraries and various browsing tools."[1]

"RAISE is an acronym for 'Rigorous Approach to Industrial Software Engineering'. It was the name of a CEC funded ESPRIT project and now gives its name to a formal specification language, the RAISE Specification Language (RSL), an associated method and a set of tools ... the tools have been commercially available for sometime". [2] It must be added that there is also a set of tools freely available from [3]

According to [4] "Formal Methods are mathematically based techniques for describing system properties. A formal method has an underlying theoretical model against which a description can be verified."

On the other hand "a formal specification language provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects satisfy each specification." [4]

Both PVS and RAISE conform to the above definitions. They are are formal languages that can be used to do formal specifications of complex software systems. And although both languages have some similarities they differ in aspects that are crucial —RSL is a much bigger language than PVS and moreover has several constructs that make the languages substantially, and in some cases, conceptually different (partial and total functions against only total ones, etc.).

The description given here includes solutions to some of these problems as well as a the details of the design of a tool that automatically translates RSL into PVS.

The Raise Specification Language (RSL) is a language suitable for formal specification and development of software systems. It supports a number of different styles of specification: abstract, property-oriented, sequential as well as concurrent specification of systems, and also allows the construction of model oriented designs.

As we said above RAISE is supported by several tools some of which are commercially available

and not in the public domain. On the other hand, there are a number of tools that can be freely accessed and that can help in the development process when using RSL. These tools are developed and maintained at UNU/IIST, see [5], but the set of these tools does not include a prover.

PVS has a potent prover that is freely available, so translating RSL into PVS has —in this case— the added benefit of producing a tool that can be freely available and would allow a specification written in RSL to be proved in PVS using the PVS prover.

The rest of this report is organized as follows. In Part I we group some semantic considerations and common characteristics of both languages: Section 2 deals with the main differences in the semantics between the two languages. Section 3 delves into the similarities and differences between common data types in the two languages. Part II groups the sections dealing with the major RSL language constructs and how and why they are mapped into a corresponding PVS construct: Section 4 shows how the modular structure of RSL is treated; Section 5 addresses issues related to the declaration structures of RSL; In Section 6 the expressions in RSL and PVS are considered and also how we can go from one to the other; in Section 7 a few syntactic issues are dealt with; in Section 8 there are some notes on the translator tool. Finally, in Section 9 some conclusions are extracted from the work presented and some future work is suggested. In Part III there are some Appendices: Appendix A has two cross reference tables that relate the different chapters of "The RAISE Specification Language" book [6], with their treatment in this report. Appendix B has tables of the RSL and PVS operators their associativity, precedence and relation to each other.

A word on the Notation. Throughout the report the following conventions are used:

- Roman font is used normally for RSL and **bold** for its keywords.
- `typewriter` font is used normally for PVS and with `UPPER CASE` for its keywords.
- common identifiers like f and `f` and term variables like value_expr and *Expr* typically indicate that the former in RSL is transformed to the latter in PVS.

The notation for the RSL syntax is taken from the RSL Concrete Grammar ([6, Appendices A, B, C, pages 371-385]) and for PVS Concrete Grammar([1, Appendix A, pages 75-82]). We have taken some liberties with this notation trying to approximate it to a more abstract grammar when necessary, but in those cases its meaning, we hope, will be clear from the context.

RSL Construction *n.m* and **PVS Construction** *n.m* are used to precede and a □ to end, an RSL or PVS language syntax form when it might become useful to referenced them. *n.m* are digits used as references to the construct.

TCC stands for Type Correctness Condition. TCCs are conditions generated by the PVS Type Checking Tool. They are proof obligations that can be discharged in the PVS system using the

PVS Prover.

Part I

General Issues

Summary. We discussed in this Part issues having to do with the semantics of the two languages and their common data types.

In Section 2, page 5 we discussed issues regarding the differences in the semantics of the languages and problems arising from them and proposed solutions to the problems.

In Section 3, page 13 we give an overview of the data types in RSL and PVS, their characteristics, similarities and differences.

2 Different Semantic Frameworks

RSL and PVS have different semantic frameworks. Although both are formal languages and their aim is to provide a tool for specifications and proof, RSL is a far larger language than PVS, and this is so not only syntactically but also in their associated semantics.

Also, even if they share some basic theoretical foundations, they are not the same. Both are specification languages based on high order logic, but RSL includes provisions for concurrency and imperative (state) style of specifications while PVS has no such constructs.

They have as well other semantic differences which must be addressed in the process of producing a transformation from one to the other. We can illustrate these by figure 1 where the whole RSL space style of specification and their relation to each other is illustrated.

As we can see there are two major divisions: Applicative and Imperative and both can be Concurrent or Sequential. We are not going to treat Concurrent and Imperative. So we are concerned only with the Applicative Sequential area, named *A-S* in the figure.

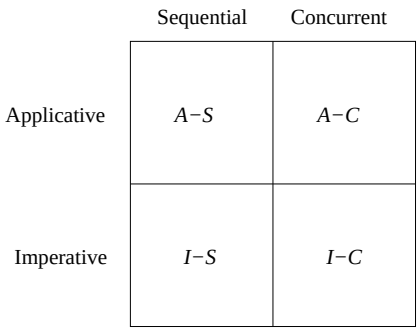

Figure 1: RSL Styles

2.1 Convergence

There is a fundamental problem in the semantics of PVS in that it only allows *convergent* expressions, expressions that are terminating and deterministic. RSL allows expressions that may be non-terminating and/or non-deterministic.

We can illustrate the problem by dividing the area *A-S* in Figure 1 into three regions as shown in Figure 2. The Convergent expressions includes literals like **true** and 1, and expressions like 1 + 0. The Non-convergent expressions include expressions like **chaos** and 1 ⌈⌉ 2.

We can avoid trying to translate the Non-convergent expressions, since we can detect them syntactically. Unfortunately, however, there is a third region of expressions for which convergence is non-decidable. For example, for arbitrary expression e1, e2, e3 and e4 the expression e1/e2 may be non-terminating if e2 is zero, and the enumerated map [e1 ↦ e2, e3 ↦ e4] may be non-deterministic on application if e1 and e3 give the same value while e2 and e4 give different values.

Such conditions for assurance of convergence are not decidable: they can in general only be

checked by proof. At the same time, if we generated translations into PVS, i.e. into necessarily convergent expressions, for expressions that are in fact non-convergent, we would be changing the theory of the specification being translated.

In the main, we can keep ourselves in the Convergent area by similar means to those used by PVS itself, such as the use of subtypes like the non-zero integers on which division may be defined as a total function. So we have particular techniques for dealing with map application, partial functions and recursive functions, as we describe later.

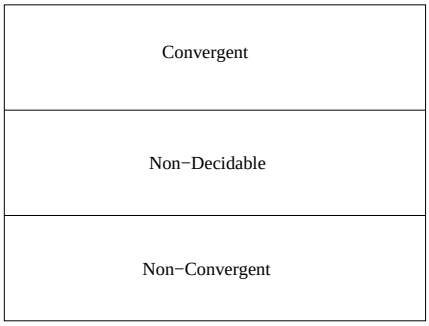

Figure 2: Convergence and Non-Convergence

But there are still some areas where more is needed:

- Enumerated maps may be non-deterministic when applied, as in the example above.
- Patterns in let expressions and cases may be inadequate to match the expressions applied to them: patterns in RSL are more general than those in PVS. This means that some translated expressions could, in RSL, be equivalent to **swap**, a form of deadlock.

We could restrict enumerated maps to empty and singletons, and restrict patterns and cases syntactically, but we prefer instead to adopt a more liberal approach of employing confidence condition generator of the RSL tool [5]. For expressions that may be non-convergent it generates confidence conditions, and we then say that the translation is only correct if the confidence conditions are satisfied. For example, the enumerated map above generates the (slightly strong) condition that e1 $\neq$ e3.

We formalise the role of confidence conditions in the next section 2.2 where we discuss correctness.

Of course, if we use our translator to try to prove confidence conditions we are in danger of circularity. But it is usually enough to check them by hand. There is a second safeguard in that many will also appear as type check conditions (TCCs) when we use PVS.

2.2 Extending the RSL Theory

Even with the treatment described in the previous section 2.1, we are extending the RSL theory a little. This, if it allows us to prove in PVS things that we could not prove in RSL, makes the translator potentially unsound.

There are two examples: application of partial functions (and operators), and application of maps.

Take the expression 1/0. In the RSL semantics this is underspecified. It could be **chaos**; it could be 1 or 0, or some other integer; it could be non-deterministic. All we know is that if we get a value it must be an integer, because of the type rules. The RSL semantics is similarly underspecified for application of a map outside its domain.

In PVS there is a stronger theory. All expressions represent values, including 1/0. And there is no non-determinism — it is always the same value.

We can summarise this by saying that in PVS the expression "e = e" is true for any expression e. In (applicative) RSL this is only so for convergent expressions. So, in translating, we are effectively adding this theorem to the theory of RSL, and so extending it.

In the case of maps we adopt a model in PVS (see section 6.3.6, page 49) that extends the theory a little more: we use a special value `nil` to represent the application of a map to a value not in its domain. This means that we have added to the RSL theory of maps a theorem that says, for any map m and values a, b not in its domain (but in its domain type), "m(a) = m(b)".

We justify these extensions to the RSL theory as follows:

- Given the semantics of PVS, we do not see how to avoid them or something close to them if we want to include partial functions and maps.
- For function applications at least, the extension will be hard if not impossible to exploit in PVS because TCCs will be generated for them. (We tried an approach for maps involving dependent types for domains, which would have generated similar TCCs, but it was very hard to use in practice in proofs.) And, of course, for applications of partial functions and maps we will get confidence conditions generated from the RSL being translated.
- Although the extension is not supported by RSL, neither is it prevented. In fact, since we are adding to the theory in an area where the semantics was previously loose, we can say that there is always a possible implementation of our original RSL that is consistent with the extension. Indeed, the reason for RSL being loose in this area is to allow for a variety of implementations.

2.3 Soundness

We explore in this section the question of the soundness of the transformation.

Given:

Definition 2.1

1. $\mathcal{S}_{RSL}$, *well formed RSL specification.*
2. $\mathcal{P}_{RSL}$, *well formed RSL proposition.*
3. $\mathcal{CC}(S_{RSL})$, *Confidence Conditions for a well_formed specification in RSL.*
4. $\mathcal{S}_{RSL} \vdash_{RSL} \mathcal{P}_{RSL}$, *valid in RSL (semantically provable).*
5. $\mathcal{S}_{PVS}$, *well formed PVS specification.*
6. $\mathcal{P}_{PVS}$, *well formed PVS proposition.*
7. $\mathcal{S}_{PVS} \vdash_{PVS} \mathcal{P}_{PVS}$, *valid in PVS (semantically provable).*
8. $\mathcal{S}_{PVS} \vdash_{t} \mathcal{P}_{PVS}$, *provable in PVS tool prover.*
9. $\mathcal{T}$, *translation tool from* $\mathcal{S}_{RSL}$ *into* $\mathcal{S}_{PVS}$ *and from* $\mathcal{P}_{RSL}$ *into* $\mathcal{P}_{PVS}$.
10. $\mathcal{T}(\mathcal{S}_{RSL}) \vdash_{t} \mathcal{T}(\mathcal{P}_{RSL})$ *means that* $\mathcal{S}_{RSL}$ *and* $\mathcal{P}_{RSL}$ *are* **accepted**, *and* $\mathcal{T}(\mathcal{P}_{RSL})$ *proved in the context of* $\mathcal{T}(\mathcal{S}_{RSL})$ *using the PVS tool.*

□

And also that an RSL specification ($\mathcal{S}_{RSL}$) is said to be *accepted* by the translator $\mathcal{T}$ if:

Definition 2.2

1. *The specification is parsed and type-checked by the RSL tool without errors or warnings, and*
2. *The specification is transformed to PVS by the translator without errors or warnings, and*
3. *The PVS file(s) produced by the translator are type-checked by the PVS tool without errors.*

□

Throughout this document we indicate the RSL constructs that are accepted or not.

It would in theory be possible to make the final check as part of the RSL translator, but only by duplicating much of the PVS tool. So while we try to check that PVS output will be well-formed, we do not guarantee it.

If an RSL specification is *accepted* and its confidence conditions are checked, we say the resulting PVS specification is *reliable*.

And assuming that the PVS tool is sound:

Definition 2.3

$$\frac{\mathcal{S}_{PVS} \vdash_{t} \mathcal{P}_{PVS}}{\mathcal{S}_{PVS} \vdash_{PVS} \mathcal{P}_{PVS}}$$

□

Our soundness rule is:

Definition 2.4

$$\frac{\mathcal{T}(\mathcal{S}_{RSL}) \vdash_t \mathcal{T}(\mathcal{P}_{RSL}), \mathcal{S}_{RSL} \vdash_{RSL} \mathcal{CC}(\mathcal{S}_{RSL})}{\mathcal{S}_{RSL} \vdash_{RSL} \mathcal{P}_{RSL}}$$

□

Where:

Definition 2.5

- $\mathcal{T}(\mathcal{S}_{RSL}) \vdash_t \mathcal{T}(\mathcal{P}_{RSL})$, *means that the translation done by* $\mathcal{T}$ *of the proposition* $\mathcal{P}_{RSL}$ *can be proved against the translation done by* $\mathcal{T}$ *of the specification* $\mathcal{S}_{RSL}$*, using the PVS Prover.*

- $\mathcal{S}_{RSL} \vdash_{RSL} \mathcal{CC}(\mathcal{S}_{RSL})$, *means that all Confidence Conditions in RSL have been checked.*

□

In summary, a proof of a *reliable* transformation is sound.

Rather than producing a formal proof of the above, we have preferred a more practical approach, namely going as systematically as possible —in the rest of this document— through every RSL language construct and showing how each can be mapped into an equivalent PVS construct or indicating where that it is not possible.

2.4 Usefulness

We need to deal with the usefulness of the transformation as an effective means of carrying out proofs of properties of RSL specifications based on transforming them to PVS.

There are a number of issues involved:

- Obviously the transformation needs to be sound, or we could not rely on our proofs. We discussed soundness in section 2.3.

- We might also like the transformation to be complete, i.e. to make it possible for anything valid in the RSL specification to be provable via the transformation. This would need (a) the PVS system itself to be complete, and (b) the transformation to be complete in the sense of both translating everything and providing translations also of sufficient RSL proof rules. We know the transformation is not complete: for reasons discussed earlier only some of RSL can be accepted. It is unlikely that PVS itself is complete: certainly no such result has been claimed for it. And it is not claimed that the RSL proof rules are complete, in fact in the form given in [2] they are admittedly incomplete.

  Completeness is therefore not a reasonable goal: what we assert instead is that:

  - We accept a large enough subset of RSL for interesting problems to be stated in it. In fact we believe our translatable subset to be as large as PVS, i.e. we could transform PVS into it, so this seems a reasonable claim based on the considerable literature on the use of PVS.
  - We think we could translate all the proof rules relevant to the accepted subset of RSL into PVS, and so be able to do with PVS all the proofs for that subset that have been done with the original RSL prover. Incompleteness has not been a practical problem with this tool.
  - Incompleteness has not, to our knowledge, been reported as a serious problem with the PVS tool.

- The original RSL proof tool has something over 200 "basic", i.e. defining, proof rules, and nearly 2000 "derived" ones. We should check that all the basic ones are provable from our translation. We expect that most will be, since we have either transformed RSL into existing PVS constructs, or else provided definitions in terms of PVS constructs. The transformation is therefore constructive rather than axiomatic, and what were originally defining rules should have become lemmas.

  The utility of the original RSL prover depended heavily on many derived rules being available: proofs done only using basic rules tend to be much longer and more tedious to construct. We expect to need fewer with PVS since its better decision procedures, and the lemmas for its existing constructs already found useful and contained in the prelude, should remove much of the need for them. But with experience we can add (after proving) those we find useful.

- Finally, we should add that the RSL tool, including the translator to PVS, is open source and available for a variety of platforms: any platform for which C can be compiled, in fact. PVS is not open source, and only runs on Sun Sparc and Linux systems, but it is free. So anyone with access to a PC, for example, can without paying anything for software set up the translator and its target environment.

2.5 The Transformation Process

The Transformation Tool $\mathcal{T}$ mentioned above have been constructed along the following lines:

Given:

($\mathcal{AG}_{\mathcal{RSL}}$) an RSL Abstract Grammar,

($\mathcal{AG}_{\mathcal{PVS}}$) a PVS Abstract Grammar

we can construct a Transformation Process , $\mathcal{T}$ that having as input an RSL Abstract Syntax Tree ($\mathcal{AST}_{\mathcal{RSL}}$) belonging to an RSL specification $\mathcal{S}_{RSL}$, will produce as output:

Definition 2.6

- ***either*** *a corresponding semantically equivalent PVS Abstract Syntax Tree ($\mathcal{AST}_{\mathcal{PVS}}$) belonging to an specification $\mathcal{S}_{PVS}$ provided that:*
  - *$\mathcal{S}_{RSL}$ is a well formed RSL specification, where by well formed we mean that the construct is not only syntactically correct but also that it is statically correct, i.e. it has been type checked and no errors have been found.*
  - *all Confidence Conditions for $\mathcal{S}_{RSL}$ have been checked.*
- ***or*** *an error if $\mathcal{S}_{RSL}$ have clear non-deterministic expressions, concurrent expressions, imperative expressions or any other construct that it is mentioned in this document as not translatable.*

□

In the rest of this report, then, we show how to map every major component of $\mathcal{AG}_{\mathcal{RSL}}$ to its possible corresponding $\mathcal{AG}_{\mathcal{PVS}}$ or when that is not possible we say so.

Of course having obtained a particular $\mathcal{AST}_{\mathcal{PVS}}$, from a $\mathcal{AST}_{\mathcal{RSL}}$, there is still the need to produce a PVS output from it that is syntactically correct, but this is a well known straightforward procedure.

2.6 Equivalence and Equality

Equivalence and Equality do not have the same meaning in RSL. Having:

$$value\_expr_1 \equiv value\_expr_2$$

and

$$value\_expr_1 = value\_expr_2$$

they will have the same meaning if and only if:

1. there are no side effects on variables,
2. both expressions are defined,
3. both expressions are deterministic

Those conditions will hold in the applicative style of specification in RSL if no partial functions, non-deterministic expressions involving $\bigsqcap$ or implicit **let** expressions are considered.

We show, in the corresponding sections in this document, how we propose to deal with partial functions and non-deterministic expressions problems as they arise, and since we only going to transform applicative RSL we can say that for our purposes both equivalence and equality in RSL have the same meaning. So RSL equivalence can be replaced by PVS equality when needed with no loss of meaning since equivalence and equality in RSL will have the same meaning if conditions 1, 2 and 3 above hold.

2.7 Under Specification

Another issue is that of Under Specification. This is accepted in PVS. Types can be declared with a minimum of assumptions about them. Functions can be declared with just their signature. Constants do not need to completely identify a value of their type.

Example 2.1 (Under Specification)
So given the following RSL:

value *x :* **Int**
axiom $x \neq 0$

the equivalent PVS will be:

```
x :  int
x :  AXIOM x /= 0
```

□

3 Theory of Common Data Types

There are several common data types shared by RSL and PVS. Apart from the traditional built-in types both languages have the capacity to declare other types.

RSL and PVS type expressions are very similar. In both cases a type is a collection of values.

In RSL there are three kind of types:

- *predefined types*: the literals built into the language.
- *abstract types*: represented by identifiers introduced in sort definitions, variant definitions, union definitions and short record definitions.
- *compound types*: built from other types by application of a *type operator* to one or more types.

How we deal with *abstract types* and *compound types* is explained in section 5, page 32. In section 3.1 we consider first the much simpler case of the *built-in types*.

3.1 Built-in Types

Both languages have a certain number of predefined or built-in types that can be used in type expressions.

PVS also has a number of types that are built into the language but it as well has a `prelude theory` that groups some definitions and declarations that can be considered for most purposes as built-in the language since they get imported into every `THEORY` by default and which our translation will use.

3.1.1 Type Literals

Type literals in RSL are: **Bool**, **Int**, **Nat**, **Real**, **Char**, **Text** and **Unit**. Each case is analysed separately below.

We deal with operators for most of the types considered below in section 7.2, page 74.

Booleans

The Boolean type in RSL is isomorphic to the PVS type `bool`, mapping **true** to `true` and **false** to `false`. In RSL equality is defined as:

$=: \mathrm{T} \times \mathrm{T} \rightarrow \mathbf{Bool}$

and the result is **true** if and only if the values of the two arguments are equal.

In PVS equality is defined with an equivalent definition as:

```
=:  [T, T -> boolean]
```

Regarding inequality, in RSL inequality is defined as:

$\neq: \mathrm{T} \times \mathrm{T} \rightarrow \mathbf{Bool}$

and the result is **true** if and only if the values of the two arguments are not equal.

While in PVS inequality is defined as:

```
/=(x, y):  boolean = NOT (x = y)
```

So both definitions are equivalent.

Numbers: (Int, Nat, Real)

PVS has a maximal type `number` that is the highest numeric type, and `real` is a subtype of `number`, `rational` is a subtype of `real`, `integer` is a subtype of `rational`, and `nat` is a subtype of `integer`.

This is not the same in RSL, where **Int** and **Real** are both different maximal types. However **Nat** is a subtype of **Int**.

This may create problems specially with some operators and we deal with them in section 7.2, page 74, where when there is a problem a possible solution is given.

There are however a number of operators in RSL that have an infix or prefix format that have a corresponding function in PVS declared in the PVS `prelude theory` so they do not have an

infix or prefix format. But this is just a syntactic matter since it does not change the semantics of the operation.

There is also a problem in PVS with real numbers since the PVS type checker does not accept real literals numbers, so a convention to introduce them has to be adopted See 6.4.1, page 52, where we explain how we propose to deal with this problem.

Characters

RSL has a built-in type **Char**, which has as its values the ASCII letters, digits as well as escape characters, octal constants (oct_constant) and hexadecimal constants (hex_constant).

There is no equivalent built-in type in PVS. But PVS defines characters as a `DATATYPE` in the `prelude theory`.

So both types are equivalent.

Text

RSL has a built-in type **Text**, representing the expression **Char***. So **Text** is a list of **Char**.

In PVS this could be the strings which are defined in the prelude as a finite sequence of `char`.

The exact definition in PVS for `string` is:

```
finite_sequence[char]
```

PVS defines sequences separately from lists, given slightly different operators to both, so RSL **Text** has some differences from PVS `string`.

However both are lists of characters and their main difference lays in the number and kind of operators available in each language.

This implies extending somehow the definition to be able to provide for PVS strings the same operations as RSL **Text** has. See section 7.2, page 74 for more information about it.

Unit

There is no **Unit** type in PVS. Functions have to be declared with at least one parameter (domain) according to the grammar rules and this must be a type.

Anyway the **Unit** type could be translated as a one element type with an arbitrary value. [2,

p. 354-5]

In any case the **Unit** type in RSL is mainly used in:

- "parameterless" functions that are mainly used for imperative rather than applicative specifications;
- for **skip** which is also useless in applicative specifications;
- and for channel values in concurrent specification which is outside of the scope of the transformation attempted here.

So **Unit** is not accepted.

3.2 SubTypes

As a type can be considered a set of values it is also natural to consider that there can be subsets of that set. This is another way of looking at subtypes, specially in the context of a language like PVS that has a tightly bound relation between subtyping and subsets, so much so that PVS considers predicate subtypes as one of "the most important feature for the PVS type system" [7]

"In RSL a type T_1 is a subtype of another type T_2 if all the values contained in T_1 are also contained in T_2". [6, page 83].

So both languages have a strong subtyping capability and in both it plays an important role. In both languages the subtypes are defined through the use of type expressions, see section 6.3, page 46.

3.3 Variants and Records

Variants play an important role and are a useful structure. They are present in both RSL and PVS. In both systems they have similar theoretical foundations.

Variants in RSL and `DATATYPE` in PVS are a mechanism provided in several languages, although in RSL and PVS —as well as other Formal Methods systems— they have additional constructs.

3.3.1 Signatures

Variant definitions in RSL are a convenient way of defining a sort, some value definitions and some axioms.

The value definitions are constructors, destructors and reconstructors of values of the sort as well as axioms defining some properties for the constructors, destructors and reconstructors.

When making a variant definition RSL automatically generates the signatures and axioms for the constructors, destructors and reconstructors that are declared in the definition.

So having the following example in RSL of a specification of a tree using a variant definition, where we choose to use Elem as a sort representing the type of each element in the tree:

```
class
  type
    Elem,
    Tree == empty | node(left: Tree, val: Elem ↔ repl_val, right: Tree)
end
```

This corresponds to the general form that variants take in RSL:

type id == variant$_1$ | ... | variant$_n$

and for each variant, indexed by i there would be a constructor con_i with the following value declaration:

value con$_i$: id

which corresponds to a constructor for a constant of type *id*, which in our example is:

value empty: Tree

or if the variant is a record variant with n_i components:

value con$_i$: (T$_{i,1}$, ... , T$_{i,n_i}$)

it would be a value declaration of the form:

value $\text{con}_i : \text{T}_{i,1} \times ... \times \text{T}_{i,n_i} \rightarrow \text{id}$

So con_i is a total function that constructs values of type *id* and which corresponds to the following function for the example given:

value node : Tree × Elem × Tree → Tree

Furthermore in this case there are as well as the three destructors: *left*, *val* and *right* one reconstructor: *repl_val*. We will deal with each one of these below.

But before going any further, let us see how the same example would look in PVS, using a `DATATYPE` definition. We are going to use an in-line definition of a `DATATYPE`.

```
Elem:  TYPE+
Tree:  DATATYPE
  BEGIN
    empty:  empty?
    node(left:  Tree, val:  Elem, right:  Tree):  nonempty?
  END Tree
```

We can say that this particular example corresponds to the general structure of a `DATATYPE` in PVS that is:

IdOp : `DATATYPE`
`BEGIN`
$\texttt{cons}_1$`(`$\texttt{acc}_{11}$`:` $\texttt{T}_{11}$`, ..., `$\texttt{acc}_{1n_1}$`:` $\texttt{T}_{1n_1}$`):` $\texttt{rec}_1$
⋮
$\texttt{cons}_m$`(`$\texttt{acc}_{m1}$`:` $\texttt{T}_{m1}$`, ..., `$\texttt{acc}_{mn_m}$`:` $\texttt{T}_{1n_m}$`):` $\texttt{rec}_m$
`END`

Where $\texttt{cons}_i$ is a *constructor* of the type, acc_{ij} is an *accesor* for the type (the destructor in RSL parlance) and rec_i a *recognizer* of the type of interest.

Then both definitions, for the example given, include the two *constructors*: `empty` and `node` and the three destructors_{RSL} or $accesors_{PVS}$: `left`, `val` and `right`.

RSL definitions also include the reconstructor called *repl_val* which does not figure in the PVS definition, since PVS does not have provisions for automatic definitions of reconstructors. These would have to be provided separately. We will show how this can be done below.

On the other hand PVS has implicit signature definitions of *recognizers*. In these case they are `empty?` and `nonempty?`.

The `Tree DATATYPE` in PVS produces the declaration of a type `Tree` and two functions corresponding to the two constructors `empty` and `node` as follows:

```
Tree:  TYPE
empty:  (empty?)
node:  [[Tree, Elem, Tree] -> (nonempty?)]
```

which correspond to the same definition in RSL, only that `node` is defined to be of type *nonempty tree*, `(nonempty?)`, and `empty` of type *empty tree*, `(empty?)`.

RSL definition include also three destructors: *left*, *val*, and *right*. These are the same in PVS only in PVS they are called *accesors*. In RSL these destructors have the following signature:

value
left: Tree $\xrightarrow{\sim}$ Tree
val: Tree $\xrightarrow{\sim}$ Elem
right: Tree $\xrightarrow{\sim}$ Tree

The destructors are partial functions in RSL since their behavior is not defined for the case of the empty tree.

In PVS these functions have the following signatures:

```
left:  [(nonempty?)  -> Tree]
val:  [(nonempty?)  -> Elem]
right:  [(nonempty?)  -> Tree]
```

We can see that both definitions are the same. The signature for each function is the same. The fact that the functions in RSL are partial is solved in PVS by having the domain of the functions be a *nonempty tree*.

We have already said that in PVS the declaration generated includes the two *recognizers* `empty?` and `nonempty?` with the following signature:

```
empty?:  [Tree -> boolean]
nonempty?:  [Tree -> boolean]
```

3.3.2 Axioms

In general every variant declaration in RSL generates axioms. The two most important are those concerning disjointness and induction.

Disjointness Axiom

In RSL the Disjointness axiom states that different constructors of the type of interest map into different values:

axiom
$\forall\ x_{i,1}: T_{i,1}, ..., x_{i,n_i}: T_{i,n_i} \bullet$
$\forall\ x_{j,1}: T_{j,1}, ..., x_{j,n_i}: T_{j,n_j} \bullet$
$con_i(x_{i,1}, ..., x_{i,n_i}) \neq con_j(x_{j,1}, ..., x_{j,n_j})$

In PVS this disjointness is enforced implicitly by a similar axiom.

Induction Axiom

Another important axiom generated by both RSL and PVS is the Induction Axiom which allows proof by induction in the variant type and in the datatype, respectively. In RSL, and for the example given, the induction axiom would be:

axiom
$\forall\ t : Tree \to \textbf{Bool} \bullet$
$($ t(empty)
$\wedge$
$(\forall\ x_1: Tree, x_2: Elem, x_3: Tree \bullet$
$(t(x_1) \wedge t(x_3)) \Rightarrow t(node(x_1, x_2, x_3)))$
$) \Rightarrow$
$(\forall\ x: Tree \bullet t(x))$

In PVS the Induction Axiom, for the `Tree DATATYPE` is:

```
Tree_induction:  AXIOM
  FORALL (t:  [Tree -> boolean]):
    (  t(empty)
       AND
       (FORALL (node1_var:  Tree, node2_var:  Elem, node3_var:  Tree):
          t(node1_var) AND t(node3_var) IMPLIES
          t(node(node1_var, node2_var, node3_var)))
    ) IMPLIES
    (FORALL (Tree_var:  Tree):  t(Tree_var));
```

Which is clearly equivalent to the RSL axiom.

Constructor/Destructor Axiom

Both RSL and PVS produce the *Constructor/Destructor* (in RSL) and *Constructor/Accessor* (in PVS) pairs of axioms.

In RSL these axioms have the following form:

axiom
$\forall$ x_1: $T_{i,1}$, ..., x_{n_i}: T_{i,n_i} •
$dest_{ij}(con_i(x_1, ..., x_{n_i}) \equiv x_j$

So for the example in question the *Constructor/Destructor* pairs of axioms in RSL would be:

axiom
$\forall$ x_1: Tree, x_2: Elem, x_3: Tree •
left(node(x_1, x_2, x_3) = x_1,
$\forall$ x_1: Tree, x_2: Elem, x_3: Tree •
val(node(x_1, x_2, x_3) = x_2,
$\forall$ x_1: Tree, x_2: Elem, x_3: Tree •
right(node(x_1, x_2, x_3) = x_3,

The corresponding axioms *Constructor/Accessor* generated in PVS are:

```
Tree_left_node:  AXIOM
  FORALL (node1_var:  Tree, node2_var:  Elem, node3_var:  Tree):
    left(node(node1_var, node2_var, node3_var)) = node1_var;
Tree_val_node:  AXIOM
```

```
    FORALL (node1_var:  Tree, node2_var:  Elem, node3_var:  Tree):
      val(node(node1_var, node2_var, node3_var)) = node2_var;
Tree_right_node:  AXIOM
    FORALL (node1_var:  Tree, node2_var:  Elem, node3_var:  Tree):
      right(node(node1_var, node2_var, node3_var)) = node3_var;
```

Which are clearly equivalent to the RSL axiom.

3.3.3 Reconstructors

RSL allows the definition of reconstructors. In the example of the tree given before this reconstructor was called *repl_val*.

Each reconstructor is introduced in a record_variant as follows:

$$\mathrm{con}_i(..., ...\ \mathrm{T}_{ij} \leftrightarrow \mathrm{recon}_{ij}, ...)$$

and it generates a **value** declaration:

$$\textbf{value}\ \mathrm{recon}_{ij} : \mathrm{T}_{ij} \times \mathrm{id} \xrightarrow{\sim} \mathrm{id}$$

So for the reconstructor *repl_val* of the tree example one would get:

$$\textbf{value}\ \mathrm{repl\_val} : \mathrm{Elem} \times \mathrm{Tree} \xrightarrow{\sim} \mathrm{Tree}$$

There are axioms associated with this declaration. For each destructor $dest_{i,k}$ associated with the variant there is an axiom relating it to the reconstructor $recon_{ij}$. If j and k are the same:

axiom
$\forall\ \mathrm{x}_j\colon \mathrm{T}_{ij}, \mathrm{x} : \mathrm{id} \bullet$
$\mathrm{dest}_{i,j}(\mathrm{recon}_{ij}(\mathrm{x}_j, \mathrm{x})) \equiv \mathrm{x}_j$

the destructor recovers the component value changed by the corresponding reconstructor. And if j and k are not the same, the axiom is:

axiom
∀ x_j: T_{ij}, x : id •
$dest_{i,k}(recon_{ij}(x_j, x)) \equiv dest_{i,k}(x)$

So in the case of the example we obtain the following axioms:

axiom
∀ x_2: Elem, x : Tree •
left(repl_val(x_2, x)) ≡ left(x),
∀ x_2: Elem, x : Tree •
val(repl_val(x_2, x)) ≡ x_2,
∀ x_2: Elem, x : Tree •
right(repl_val(x_2, x)) ≡ right(x),

As we have said before PVS does not have provisions for reconstructors. So to be completely compatible the transformation must generate independently the reconstructors and their axioms whenever needed.

In this particular case the reconstructor *repl_val* will have the following definition:

```
repl_val (e : Elem, t : (nonempty?)) : (nonempty?) =
  node(left(t), e, right(t))
```

To avoid the problem of *repl_val* being partial, `(nonempty?)` is used as domain and range of the function.

The reconstructor could also be defined with only one axiom as follows:

```
repl_val_axiom: AXIOM FORALL(x1, x3: Tree, x2, x2_1: Elem):
  repl_val(x2, node(x1, x2_1, x3)) = node(x1, x2, x3)
```

However this definition generates a TCC in PVS. Better then is to define the reconstructor as was done above.

3.3.4 Wildcards

RSL allows defining a variant with wildcard constructors '_'. This means that for this case no induction axiom is generated.

PVS does not have a similar wildcard construct. So the equivalent in PVS is according to the RSL expansion, with no induction axiom.

3.3.5 Short Record Definitions

Short Record definitions in RSL are shorthand for variant definitions that have a single variant including one constructor, so they can be transformed as a more simple instance of a PVS `DATATYPE`, where the constructor is the mk_ function implicit in RSL.

We could have used PVS records but these are not so convenient because their syntax differs quite considerably from RSL's.

Part II

Language Constructs

Summary. In this Part the major Language Constructs are dealt with.

In Section 4, page 27 we show how we propose to transform Modules.

In Section 5, page 32 we show how we propose to transform Declarations.

In Section 6, page 46 we show how we propose to transform Expressions.

In Section 7, page 73 we show how we propose to transform identifiers, operators, and some other syntactic issues.

In Section 8, page 96 there are notes on the Tool, its structure and characteristics as well as its scope.

In Section 9, page 99 there are some conclusions, a survey of related work and suggestions for future work.

Specifications. Before going into a discussion on modules and other language constructs in both languages it seems interesting to address the subject of specifications.

For PVS a specification is a collection of *theories*, i.e. modules, and in RSL they are a composition of modules [6, page 204].

So we can consider that in the context of both languages a specification is a set of one or more declarations included in one or more modules.

So a specification is not something that should worry us too much since if we can transform all the other language constructs we are going to end transforming whatever specification was intended.

Introduction. An RSL *declarative construct* represents one or more *definitions*, which in turn introduces identifiers or operators for entities: object, type, variable, value, channel or axiom.

Schemes, objects, theories, development relations are modules in RSL. The other are declarations of constructs that are inside a module.

RSL entities such as variable and channel do not have a corresponding construct in PVS and they

are not considered here since they are used in the concurrent or imperative style of specification which is not included in the transformation. In general objects and schemes are treated as modules hence they can be rendered as theories. There are more details of this in the section that deals with modules, 4, page 27.

A PVS *declaration* introduces types, variables, constants, formulas, judgments and conversions. Each *declaration* has an *identifier* and belongs to a unique theory. Declarations have a body which indicates the *kind* of the declaration, provides the signature and definition of the declaration.

RSL employs the concept of context to define the scope rules for declarative constructs. So for each declarative construct the scope may depend on the context in which it occurs.

PVS's scope is associated with the concept of `THEORY`. Each declaration has as its scope the `THEORY` in which it is declared. This is discussed further in the section on modules, 4, page 27.

In general we can say that RSL *declarative constructs* correspond roughly to *declarations* in PVS. We are going to show how these RSL *declarative constructs* can be mapped into PVS *declarations* in the following sections.

4 Modules

Modules in RSL are schemes, (global) objects, theories, and development relations. Each are defined in separate RSL files and will be translated to separate PVS files with the same base name. The content of a PVS file will be a `THEORY` of the same base name, possibly with extra theories necessary, as we shall see, for embedded objects and for class scope expressions.

Modules in RSL may depend on other modules, which appear in their context. In PVS these context references will, with the exception of parameterised schemes, be translated by a corresponding `IMPORTING` declaration in PVS.

4.1 Schemes and Objects

In the applicative case there is no essential difference between schemes and objects, and we shall take advantage of this in translating them both to PVS `THEORY`s. The body of the theory is in each case the translation of the class expression of the scheme or object: see section 4.5.

We do not currently accept parameterised objects (though they could be translated using parameterised theories) because they are not normally used except for imperative or concurrent RSL.

Parameterised schemes are not currently accepted. They are hard to translate in general since RSL allows axioms in parameter classes, and PVS does not. Instead, we translate definitions of parameterised schemes by replacing the parameters by embedded object definitions. A parameterised scheme of the form

scheme S(A: C1, B: C2) = **class** class_expr$_S$ **end**

can be transformed as if it was declared in RSL as:

scheme S =
  extend class
    object A: C1, B: C2
  end
  with class class_expr$_S$ **end**

And the equivalent PVS will be:

```
A: THEORY
  BEGIN
    transformation of C1
END A
```

```
B: THEORY
   BEGIN
      transformation of C2
END B

S: THEORY
   BEGIN
      IMPORTING A, B
      TheoryPart_S
END S
```

This means that we must translate instances of parameterised schemes in terms of their unfoldings, as we shall see in section 4.5. Instances of non-parameterised schemes can be translated by `IMPORTING`.

4.2 Theories and Development Relations

Theories and development relations involve axioms in RSL, and are translated into PVS theories containing corresponding `LEMMA`s. They are not really axioms, as they are things to be proved.

Theories and development relations may involve the special logical expressions the class_scope_expression, the implementation_relation and the implementation_expression.

The class scope expression

in C ⊢ E

is translated into a small PVS theory, containing the declarations of the class expression C plus the translation of E as a lemma. We use a theory for each class scope expression because in an RSL theory there may be several such expressions with possibly different class expressions, and we need to limit the scope of the class's declarations.

Implementation relations and implementation expressions are translated according to their expansions, generated by the RSL tool. The expansions generate class scope expressions.

4.3 Object Declarations

As well as the global objects already discussed, objects can be declared inside class expressions. They would naturally give rise to PVS theories nested inside the PVS theory of their class expression. But this is not allowed in PVS, so we need to "flatten" them. For example, consider the following RSL scheme definition:

scheme S =
  class
    object A :
      class
        object B : **class** class_expr$_B$ **end**
          class_expr$_A$
      end
      class_expr$_S$
  end

This would be the equivalent PVS:

```
B : THEORY
  BEGIN
       TheoryPart_B
END B

A : THEORY
  BEGIN
       IMPORTING B
       TheoryPart_A
END A

S : THEORY
  BEGIN
       IMPORTING A
       TheoryPart_S
    END S
```

This means that we do not accept object classes that reference entities in their context. For example class_expr$_A$ cannot refer to anything in class_expr$_S$.

4.4 Object Expressions

Object expressions appear in two ways in RSL: as actual scheme parameters and as qualifiers in names. The first does not arise, since scheme instantiations are unfolded.

Since object declarations are "flattened", as we described in section 4.3, nested qualifications will be translated as the innermost qualification: A.B.f in RSL will be `B.f` in PVS. element_object_expressions and array_object_expressions are not accepted, as object arrays are not, and fitting_object_expressions are not accepted as qualifiers: in practice they are only used in scheme instantiations.

4.5 Class Expressions

basic_class_expressions are translated as their declarations: see section 5, page 32.

extending_class_expressions are translated as their constituent class expressions, the second appended to the first.

hiding_class_expressions are not currently accepted, but could be translated using PVS' `EXPORTING ALL BUT` construct. Not hiding names may possibly lead to name clashes in the translated PVS.

renaming_class_expressions are accepted, but the renaming is effectively ignored, as there is no renaming facility in PVS. For example:

extend use a **for** b **in class value** b : **Int end with** ... a ...

is translated as if it had been written

extend class value b : **Int end with** ... b ...

This means that there may be name clashes in the translated PVS, so a warning is given by the translator.

with_class_expressions are translated with the **with** part ignored: all names will be qualified in the translation.

scheme_instantiations are translated using `IMPORTING` if the scheme has no parameters, as the unfolded scheme body otherwise.

4.6 Using Parameterised Schemes

The unfolding of parameterised schemes means that, if you have several class scope expressions involving the instantiation of the same scheme, even with the same parameters, they will appear as different theories to PVS, and lemmas proved in one such theory will not be available in others.

To overcome this it will be better to define the actual parameters as global objects, and to make a new scheme definition of an non-parameterised scheme whose body is the original scheme instantiation. Then each class scope expression can be written instantiating the new, non-parameterised scheme in its class scope expression, and will be translated as a theory involving

the `IMPORTING` of the same PVS theory. Then each succeeding theory can be changed in PVS to import, instead of the theory corresponding to the non-parameterised scheme, its enrichment which is the previous theory.

For example, if we have originally

scheme S(X : C1) = C2

and in an RSL theory

[l1] **in** S(O) ⊢ e1
[l2] **in** S(O) ⊢ e2

where the class of object O implements class expression C1, then we

1. Make a global object O
2. Make a scheme definition in a new module:

   scheme S_O = S(O)

3. Rewrite the RSL theory

   [l1] **in** S_O ⊢ e1
   [l2] **in** S_O ⊢ e2

4. Translate the RSL theory getting

```
l1 : THEORY
  BEGIN
    IMPORTING S_O;
    l1 : LEMMA e1;
  END l1

l2 : THEORY
  BEGIN
    IMPORTING S_O;
    l2 : LEMMA e2;
  END l2
```

5. Change the `IMPORTING S_O` in the second theory to `IMPORTING l1`

The final step is also possible and useful in the case where the original class scope expressions involved the same non-parameterised scheme.

5 Declarations

5.1 Introduction

We aim in this section to deal with declarations in RSL and how they can be mapped into a corresponding PVS construct that is semantically equivalent.

Declarations are the way in PVS and RSL of introducing the major different entities in both languages.

Declarations in RSL are object declaration, type declaration, value declaration, variable declaration, channel declaration, axiom declaration and test case declaration. Each of these RSL declarations is treated separately below.

5.2 Object Declarations

An Object Declaration in RSL "is essentially a named model chosen from a class of models represented by some class expression " [6, page 205].

object_declaration ::= **object** object_def-list
object_def ::=
opt-comment_string
id opt-formal_array_parameter : class_expr
formal_array_parameter ::= [typing-list]

Since Objects are closely related to the concept of module in RSL we refer the reader to the Section on Modules: Section 4, page 27.

5.3 Type Declarations

A Type Declaration in RSL is a collection of zero or more Type Definitions, introduced with the word **type**.

Type Definitions are a way of naming types, either built-in types or types declared by the user by associating a name to a type expression.

There are different kinds of Type Definitions in RSL:

type_decl ::= **type** type_def-list
type_def ::=
| sort_def

| variant_def
| union_def
| short_record_def
| abbreviation_def

In PVS Type Declarations are also used to introduce new type names associated with a type expression.

There are four kinds of type declarations in PVS:

- *uninterpreted type declaration*
- *uninterpreted subtype declaration*
- *interpreted type declaration*
- *enumeration type declaration*

We can say that in general RSL **type** definitions corresponds to `TYPE` declarations in PVS. We treat each instance of an RSL **type** definition below.

In the following discussions about type declarations, whenever we postulate that:

$\mathsf{type\_def}_{RSL} \equiv_{\mathcal{T}} \mathit{TypeDecl}_{PVS}$

and when in a $\mathsf{type\_def}_{RSL}$ is included a type expression ($\mathsf{type\_expr}_{RSL}$), and in a $\mathit{TypeDecl}_{PVS}$ is included a type expression ($\mathit{TypeExpr}_{PVS}$), we assume that:

$\mathsf{type\_expr}_{RSL} \equiv_{\mathcal{T}} \mathit{TypeExpr}_{PVS}$

This last equivalence is discussed in section 6.3, page 46, for each RSL type expression.

5.3.1 Sort Definitions

In RSL Sort Definitions are a way of introducing an abstract type. This is equivalent to uninterpreted type declarations in PVS, which are a way of declaring a type with a minimum of assumptions about it.

Of course, in this case —and for both languages— there is no type expression associated with the declaration, and they do not have any operator other than the equality operator.

Empty and Nonempty Types. However PVS distinguishes between empty (defined with the reserved word `TYPE`) and nonempty types (defined with the reserved word `TYPE+`). This distinction does not exist in RSL and can be important in the transformation since PVS type checker will generate an *existence* TCC for every constant declared if it can not determine if the type of the constant is a nonempty type. This is so since PVS considers inconsistent the existence of a constant of a type that is possibly empty.

So we propose to transform every RSL sort into a PVS nonempty type (`TYPE+`). But this extends somewhat the transformation since is like adding in RSL:

axiom ∃ t: T • **true**

This is unlikely to cause any problems in extending the RSL theory since the declaration of a value of type T or a total function with non-empty domain type and result type T would also imply it. Since it avoids getting extra TCCs we adopt it.

5.3.2 Abbreviation Definitions

These definitions, in RSL, introduce an identifier for the type represented by a corresponding type expression. These are the same as the *interpreted type declaration* of PVS that are also a means of given a name to a type expression.

5.3.3 Short Record and Variant Definitions

Short Record and Variant Definitions in RSL introduce an identifier for the type expression defining a variant. Variants are treated in section 3.3, page 16.

5.3.4 Union Definitions

Union Definitions in RSL are a short hand to define a special class of variants. They could be dealt with by:

a. Expanding the union definition with its equivalent variant definition.

b. Making explicit any implicit coercions in the RSL code.

Translations of Union definitions are not accepted in the tool currently.

5.4 Value Declarations

Values of types in RSL can be given a name through a Value Declaration.

value
value_definition$_1$,
⋮
value_definition$_n$,

Values of types can also be given names in PVS, only in PVS these are considered as *Constant Declarations*.

A Value Definition in RSL is one of the following:

- typing
- explicit value definition
- implicit value definition
- explicit function definition
- implicit function definition

5.4.1 Typing

Typing value declarations in RSL are a way of defining a value of a particular type and has the form:

RSL Construction 5.1
binding : type_expr
where a binding is one or more identifiers.

□

So this is a way of binding an identifier to a type expression (type_expr).

This corresponds to a *ConstantDecl* in PVS that also binds one or more identifiers to a type expression (*TypeExpr*):

PVS Construction 5.1
IdOp : TypeExpr

□

5.4.2 Explicit Value Definitions

An Explicit Value Definition in RSL bind one or more identifiers to a type expression and assigns a particular value to them. It has the form:

RSL Construction 5.2
binding *: type_expr = pure-value_expr*

□

In PVS the corresponding declaration has the form:

PVS Construction 5.2
Identifier : *Type Expression* = *Expression*

□

5.4.3 Implicit Value Definitions

Implicit Value Definitions in RSL are a short hand for a value definition and an axiom. They have the form:

binding : type expression • value expression.

And this is short for:

RSL Construction 5.3
value *binding : type expression*
axiom *value expression*

□

It must be remarked that the value expression is a *pure, logical* one, which means that the maximal type must be **Bool** and must not read or write to variables.

There is not a directly comparable declaration in PVS for the shorthand version, however a RSL Implicit Value Definition can be rendered in PVS considering its longer form: a value definition and an axiom.

So in PVS this will become:

PVS Construction 5.3
Identifier : Type Expression
Identifier : `AXIOM` *Expression*

□

5.5 Function Definitions

Functions definitions are value declarations in RSL but we are going to discuss here the way to deal with them specially since they present one of the main issues concerning semantics in the transformation as was pointed out in section 2.1, page 5.

There are several styles of defining a function in RSL —predicative, algorithmic, algebraic— but these are just styles of defining a function and they all boil down to two basic definitions constructions in the language that encompass all of them: Explicit Function Definitions and Implicit Function Definitions.

All the definition styles mentioned above can be obtained using one or the other already mentioned language constructs.

So if we can find a way to satisfactory rendered in PVS both language constructs we are able to say that we can cover all the definitions styles of defining a function in RSL.

Of course there is a major obstacle to overcome here and that is the fact that RSL includes partial functions in its function space while PVS function space is restricted to total functions. We deal with this below, in section 5.5.3.

We also reserve a section down below to discuss the issue of recursive functions in RSL and PVS which we think deserve a special treatment, in section 5.5.4.

5.5.1 Explicit Function Definitions

An Explicit Function Definition of a Total Function in RSL take the form:

RSL Construction 5.4

binding : type_expr
formal_function_application ≡ value_expr
where type_expr *is the function signature, and so is:*
type_expr$_{functiondomain}$ → type_expr$_{functionresult}$
formal_function_application *is the identifier of the function and its formal parameters and* value_expr *is the function definition.*

□

We are purposefully ignoring here the optional pre_condition since this would make it a partial function and we deal with partial functions specially in section 5.5.3.

This definition is equivalent syntactically and semantically to the corresponding PVS construct:

PVS Construction 5.4

$IdOp(Bindings_{functiondomain})$: $TypeExpr_{functionresult}$ = $Expr$

where:
$\mathcal{T}$(type_expr) $\equiv \{Bindings_{functiondomain},\ TypeExpr_{functionresult}\}$
$\mathcal{T}$(value_expr) $\equiv Expr$

and where Bindings in PVS is:
$Bindings ::= [IdOp:\ TypeExpr]_{++}$

□

A concrete example of the above transformation would be:

Example 5.1
So given the following RSL:

value
f: **int** → **int**
f(*x*) ≡ **if** *x* ≥ *0* **then** *x* **else** -*x* **end**

the equivalent PVS will be:

```
f(x:int):  int = IF x >= 0 THEN x ELSE -x ENDIF
```

□

5.5.2 Implicit Function Definitions

Implicit Function Definitions in RSL are of the form:

binding : type_expr
post_condition
opt-pre_condition

where the type_expr is the function signature, followed by the formal_function_application and the function definition that is a post_condition that has an optional binding and a *readonly logical* value_expr.

All this must be read as follows: the post_condition is the implicit definition of the function, the binding in it has the type of the function range, and can be used in the value_expr that actually defines the function by given the condition that must be fulfilled by the result.

Actually the whole Implicit Function Definition is a shorthand for a function signature declaration and an axiom that stands for the function definition, using a **post** expression. So the axiom has the following form:

axiom
value_expr$_1$ **as** binding
post value_expr$_2$
opt-pre_condition

And this is equivalent to:

RSL Construction 5.5

opt-pre_condition ⇒
(∃ binding: type_expr •
binding = value_expr$_1$ ∧ value_expr$_2$)

□

provided that the value_exprs are applicative and terminate with a unique result and the type of value_expr$_1$ is type_expr and binding captures no free names in value_expr$_1$.

Another way of writing the above could be:

RSL Construction 5.6

pre_condition ⇒
let binding = value_expr$_1$ **in** value_expr$_2$ **end**

□

Which could be transformed into PVS as:

PVS Construction 5.5

```
pre_condition IMPLIES
LET Binding =  Expr1
IN  Expr2
```

□

However, RSL Construction 5.5 gives us an inkling as to another way we can transform the RSL into PVS. We can use *Identifier* : *Type Expression* —that is the function signature—, and an axiom that can have the same form as would the RSL axiom (see section 5.6, page 45).

So provided we have:

$\mathcal{T}(\text{binding}_{RSL}) \equiv Binding_{PVS}$
$\mathcal{T}(\text{type\_expr}_{RSL}) \equiv TypeExpr_{PVS}$
$\mathcal{T}(\text{value\_expr}_{1_{RSL}}) \equiv Expr_{1_{PVS}}$
$\mathcal{T}(\text{value\_expr}_{2_{RSL}}) \equiv Expr_{2_{PVS}}$

If we consider only the RSL **post** expression in the RSL Construction 5.5, this can be transformed into PVS as:

PVS Construction 5.6

```
EXISTS (Binding:  TypeExpr):
      Binding =  Expr1 AND Expr2
```

□

Let us see a concrete example of the above:

Example 5.2
So given the following RSL:

value
f: **int** → **int**
f(x) **as** *y*
post $y * y = 4$

the equivalent PVS will be:

```
f:  [int -> int]
f:  AXIOM FORALL (x:int) :
     EXISTS (y:  int):
          y = f(x) AND y * y = 4
```

□

We have again left aside the case of partial implicit function definitions, those with an pre_condition, since they are discussed separately in section 5.5.3.

5.5.3 Partial Functions

Since the PVS function space does not include partial functions while the RSL function space does there must be a way to transform partial functions defined in RSL to total functions in PVS.

Of course an easy solution will be to restrict the transformation to a function space that only has total functions.

We thought this too much restrictive and therefore try to find a convenient way to do an efficient automatic transformation from partial to total functions.

In the next sections, we will explore a few alternative solutions to this problem, even some that may seem promising but that are easily seen to be incorrect.

Using a special value

In this instance we analyse the situation where a new value is added to the range of the function.

For example given the following RSL definition:

RSL Construction 5.7
$f : T \xrightarrow{\sim} T$
$f(x) \equiv \mathcal{E}(x)$
pre $\mathcal{E}_p(x)$

□

One possible PVS equivalent could be to transform the RSL predicate $\mathcal{E}_p$(x) into a PVS Boolean function and used it as follows:

```
f(x:  T): T1 = IF Ep(x) THEN E(x) ELSE nil ENDIF
```

This of course would mean that the PVS function `f` will return a special value — in this case `nil`— that was not intended in the original RSL definition. The type of the PVS function will

not longer be `T` as in the RSL definition but a new type `T1` that would have to include the new special value `nil`.

Using PVS Subtypes

Another way to solve the problem would be using subtypes, such as can be done in RSL.

So again, given:

RSL Construction 5.8
$f : T \xrightarrow{\sim} T$
$f(x) \equiv \mathcal{E}(x)$
pre $\mathcal{E}_p(x)$

□

A PVS subtype can be defined:

```
st:  TYPE = { x:  T | Ep(x)}
```

And now `f` has a different signature:

```
f(x:  st):  T = E(x)
```

Anyway the maximal type continues to be same which is all right.

Using PVS Dependent types

Another alternative is employing PVS Dependent types.

Again using the same RSL:

RSL Construction 5.9
$f : T \xrightarrow{\sim} T$
$f(x) \equiv \mathcal{E}(x)$
pre $\mathcal{E}_p(x)$

□

We can define the following PVS dependent type:

```
e1(x:  T) : bool
e:  T
tf:  TYPE = [st:  pred[T], [(st) -> T]]
```

and function f can now be defined in PVS as:

```
f:  tf = (e1, LAMBDA(y:(e1)):  e)
```

But now the function is a pair: the function itself plus the predicate for the dependent type and this can create awkward problems in application since the function type is different.

Using Axioms

Still another option for the problem of partial function is to use axioms in PVS.

So given the partial function f in RSL:

RSL Construction 5.10
$f : T \xrightarrow{\sim} T$
$f(x) \equiv \mathcal{E}(x)$
pre $\mathcal{E}_p(x)$

□

we can transform it into the following declaration in PVS that involves a constant declaration:

```
f:  [T -> T]
```

and axiom such as:

```
f_ax:  AXIOM FORALL (x:  T):
       Ep(x) IMPLIES f(x) = E(x)
```

However the PVS construct is stronger than RSL's, since unless T is a maximal type in RSL, the partial function f in RSL *always* returns a value, even when the precondition is false. By defining the function in PVS using an axiom we are effectively limiting the function space to a total space.

We think this method is better suited to the implementation needs since it also deals with the recursive case. See 5.5.4, page 44.

Summary

So we can summarize the above saying that partial functions in RSL can be transformed in PVS according to the following rules:

An Explicit Function Definition that has the form:

RSL Construction 5.11
binding : type_expr
formal_function_application value_expr
optional pre_condition

□

can be transformed into the following PVS:

PVS Construction 5.7
Identifier (Bindings$_{domain}$*) : TypeExpr*$_{result}$ = *Expr*

□

Provided there is no pre_condition or recursion, otherwise it will be in PVS:

PVS Construction 5.8
Identifier TypeExpr$_{signature}$
Identifier `AXIOM`
Expr$_{precondition}$ `IMPLIES` *Expr*

□

An Implicit Function Definition that has the form:

RSL Construction 5.12
id : type_expr
post_condition
optional pre_condition

□

And we already know that the post_condition and the optional pre_condition are actually:

RSL Construction 5.13
pre_condition ⇒
∃ *id:* type_expr •
id = value_expr$_1$ ∧ value_expr$_2$

□

So this can be rendered as the following PVS equivalent:

PVS Construction 5.9
Identifier : Type Expression$_{signature}$
Identifier : `AXIOM FORALL (`*Bindings*$_{domain}$`) :`
Expr$_{precondition}$ `IMPLIES`
`EXISTS (id:` *Type Expression*$_{result}$`) :`
`id =` *Expr*$_1$ `AND` *Expr*$_2$

□

So we have ways of transforming every kind of non-recursive function definition in RSL into a corresponding equivalent in PVS. This gives us the means of effectively translating to PVS all the styles that are used in RSL, as we saw above.

5.5.4 Recursive Functions

Since the PVS function space is restricted to total functions only, to make sure that a recursive function really terminates PVS requires that all recursive definitions are provided with a `MEASURE` and an optional order relation to insure that the function remains total.

This implies that one way of transforming a recursive function in RSL —that does not have PVS restriction regarding non-terminating functions— is to find the corresponding `MEASURE` for a given RSL recursive function.

This is a rather daunting task since it is in general impossible to find an algorithm that produces the correct `MEASURE` for a recursive function.

We have opted for a simpler solution namely defining the recursive functions through axioms. This also solves the problem of the mutually recursive functions which are not permitted in PVS.

So given the following definition in RSL:

RSL Construction 5.14
$f : T_1 \rightarrow T_2$
$f(x) \equiv \mathcal{E}(f(x))$

□

A possible solution to avoid needing a measure function in PVS then is:

PVS Construction 5.10

```
f:  [T1 -> T2]
f_axiom:  AXIOM FORALL (x:  T1):
   f(x) = E(f(x))
```

□

A concrete example then would be:

Example 5.3
So given the following RSL:

f: **Int** $\rightarrow$ **Int**
$f(x) \equiv$
  if $x > 0$
    then $f(x - 1) * x$
    else 1
  end

the equivalent PVS will be:

```
f :  FUNCTION [int -> int]
```

```
f :  AXIOM FORALL (x:  int):
     f(x) =
     IF x > 0
       THEN f(x - 1) * x
       ELSE 1
     ENDIF
```

□

Preconditions in recursive functions are dealt with as for partial functions discussed in section 5.5.3, page 40.

5.6 Axiom Declarations

Axioms Declarations in RSL are a list of:

RSL Construction 5.15
[*optional* identifier] *readonly logical* value_expr

□

and are a way of stating properties of values.

In PVS a Formula Declaration is:

PVS Construction 5.11
Identifiers : Formula Name Expr, where Formula Name can be `AXIOM`

□

Here as well they are used to express properties about constants —that are the PVS equivalent to RSL values.

In both cases the expression that constitutes the body of the axiom must be a boolean expression. So if we have value_expr$_{RSL}$ $\equiv_{\mathcal{T}}$ *Expr*$_{PVS}$ (this is discussed in section 6.4, page 51), we can transform RSL Axioms Declarations into PVS *Formula Declarations*.

Of course since an Axiom Declaration in RSL is a list of *readonly logical* value_expr while a *Formula Declaration* in PVS only takes one *Expr*, the transformation here implies having as many PVS *Formula Declarations* as value_expr are in the RSL Axioms Declaration.

5.7 Variable and Channel Declarations

Since the transformation from RSL into PVS does not include either the concurrent part of RSL or the imperative one, channels and variables are not accepted.

6 Expressions

We discuss in this section the different kinds of RSL expressions and show their mapping into the equivalent PVS expressions.

There are in RSL expressions for Class, Object, Type and Value, each is discussed separately below.

6.1 Class Expressions

Class Expressions in RSL can be basic_class_expr, extending_class_expr, hiding_class_expr, renaming_class_expr or scheme_instantiation.

Since class expressions in RSL are the kernel module concept for modules we treated class expressions in Section 5, page 32.

6.2 Object Expressions

An Object Expression in RSL represents an object. The specific object is defined as an object declaration, consequently, and since there is no concept of objects in PVS, we refer the reader to the Sections on Object Declarations 5.2, page 32 and specially Modules (Section 4, page 27) where we show how we treat objects.

6.3 Type Expressions

Type Expressions in RSL can be: Literal and this has been discussed in 3.1.1, page 13; Function; Product; Set; List; Map; Subtype; and Bracketed, and all these are discussed below.

6.3.1 Names

In RSL when a type_expr is a name this is a way of representing a type that has already been defined as a type through a Type Definition. PVS has exactly the same construct for a Type Expression, so definitions are equivalent.

6.3.2 Function Type Expressions

In RSL a Function Type Expression is of the form:

RSL Construction 6.1
function_type_expr ::= type_expr function_arrow result_desc

□

The construct function_arrow just serves to indicate whether the function is partial or total.

Since is only the signature of the function that is defined here, we can ignore the difference between a partial and a total function at this level and render all RSL function type expressions as total functions in PVS.

We can reserve the treatment of partial functions, to when more than their signature definition is given as when the function is defined as a value definition. This is discussed in section 5.5.3, page 40.

The result_desc construct is a just a type_expr, if we exclude the access_desc construct that falls outside the scope of the transformation, since it refers to the imperative part of the RSL language.

So in PVS the RSL definition is equivalent to:

PVS Construction 6.1
FunctionType ::=
[{ [*IdOp* :[*TypeExpr* }$_{++}$ ',' -> *TypeExpr*]

□

Which is equivalent to the RSL definition.

Example 6.1
Given the following RSL:

type
f: T1 → T2,

the equivalent PVS will be:

```
f :  [T1 -> T2]
```

□

6.3.3 Product Type Expressions

Product type expressions in RSL have the form:

RSL Construction 6.2
type_expr$_1$ × ... × type_exprtype_expr$_n$
for $n \geq 2$

□

and they represent Cartesian products.

PVS has similar constructs called tuples which have the form:

PVS Construction 6.2
`[`t_1`, ... `t_n`]`
where $n \geq 1$

□

and they also represent Cartesian products. So all the RSL products can be transformed into PVS.

6.3.4 Set Type Expressions

Sets in RSL are a collection of distinct and unordered values of some type.

Sets are defined in PVS Prelude as predicates: `[t -> bool]`. There is a separate theory for finite sets which are defined as a subtype of set.

So RSL Set Type Expressions of the form:

type_expr**-set**

can be transformed into PVS as:

`finite_set[`*TypeExpr*`]`

and

type_expr**-infset**

into

`setof[`*TypeExpr*`]`

Example 6.2
Given the following RSL:

$f = T \to (P \times P)$**-set**

the equivalent PVS will be:

```
f:  [T -> finite_set[[P, P]]]
```

□

6.3.5 List Type Expressions

In RSL a List is a sequence of values of the same type. RSL has finite and infinite lists.

In PVS Lists are define in the Prelude as a `DATATYPE`. There is a constructor i.e.: `(:a, b:)`, and operators defined in the Prelude.

There are no definitions in the Prelude for infinite lists but there is a definition of `sequences` that are infinite.

So RSL Finite List Expressions of the form:

type_expr$*$

can be transformed into PVS as:

`list[`*TypeExpr*`]`

RSL Infinite Lists are not accepted.

6.3.6 Map Type Expressions

In PVS there are no maps as they are defined in RSL. We are going to define maps on a separate `THEORY` in PVS, and we are going to do it using a PVS function definition and a `DATATYPE`. The domain of the RSL map is the domain of the map function in PVS, while the range of the RSL map is a PVS `DATATYPE`.

So our definition is as follows. First we define the map range as a `DATATYPE`:

PVS Construction 6.3

```
Maprange[rng: TYPE]: DATATYPE
BEGIN
  nil: nil?
  mk_rng(rng_part: rng): nonnil?
END Maprange
```

□

This is the definition of the map theory, the `dom` operator and additional theorems:

PVS Construction 6.4

```
Map[map_dom: TYPE+, map_rng: TYPE+] : THEORY
BEGIN
  IMPORTING Maprange[map_rng]
```

```
map:  TYPE+ = [map_dom -> Maprange]

dom(m:  map) :  set[map_dom] =
   (LAMBDA(d:  map_dom) :  nonnil?(m(d)))

mapdef :  THEOREM
   FORALL (m :  map, d :  map_dom) :
   member(d, dom(m)) IMPLIES nonnil?(m(d))

mapdef_2 :  THEOREM
   FORALL (m :  map, d :  map_dom) :
   (EXISTS (r :  map_rng) :  r = rng_part(m(d))) WHEN nonnil?(m(d))
```

□

In the same `Map THEORY` we have the definitions of all map operations that are the corresponding transformations for the RSL map operators, and some additional ones that are used as support. The RSL operators are transformed as functions in PVS, and to make the latter compatible with those of RSL, their domain and range are the same as those of the corresponding RSL operator. We explain the transformation for each of them in the section on operators: 7.2, page 74.

As we defined maps as functions in PVS they are potentially infinite and they are deterministic. Non-deterministic RSL maps are not accepted.

6.3.7 Subtype Expressions

In RSL a Subtype Expression has the following syntax:

subtype_expr ::= { | single_typing *pure*-restriction | }
restriction ::= • *readonly_logical*-value_expr

And in PVS a Subtype Expression is:

Subtype ::= { *SetBindings* | *Expr* } | (*Expr*)
SetBindings ::= *SetBinding* [[,] *SetBindings*]
SetBinding ::= { *IdOp* [:*TypeExpr*] } | *Bindings*

The *Expr* following the list of *SetBinding* gives the equivalent condition to the RSL restriction. So both expressions are equivalent.

6.3.8 Bracketed Type Expressions

In RSL a Bracketed Type Expression is:

bracketed_type_expr::= (type_expr)

and it represents the same type as represented by the type_expr.

In PVS this corresponds to a tuple type containing only one type.

6.3.9 Access Descriptions

Access Descriptions in RSL are used to describe type of access in a function when using the imperative style of specification and since this is outside the scope of the transformation we are not concerned with them.

6.4 Value Expressions

The effect of value expressions in RSL on a state will be ignored since this relates to variables or channels that are used in the concurrent or imperative style of specification and both are outside the scope of the transformation.

Value expressions in RSL correspond roughly to the Expression (*Expr*) construct in PVS.

We consider below all RSL value expressions identifying those for which a transformation ($\mathcal{T}(\mathsf{value\_expr}_{RSL}) \equiv Expr_{PVS}$) is possible and those for which there is no possible transformation.

For those falling in the first group, we give the proposed transformation and where appropriate we also give the constraints that are imposed on the transformation.

6.4.1 Value Literals

Value Literals in RSL and their equivalent in PVS are:

- **Unit**

  unit_lit::= ()

  The **Unit** literal it is not needed for the transformation. See also section 3.1.1, page 15.

- **Bool**

  bool_lit::= **true** | **false**

  This is equivalent in PVS where `FALSE, TRUE` are defined as constants of type bool:

  ```
  boolean:  NONEMPTY_TYPE
  bool:  NONEMPTY_TYPE = boolean
  ```

`FALSE, TRUE: bool`

- **Numbers**

  See also section 3.1.1, page 14.

  - Integers

    In RSL integers literals are defined as:

    int_lit::= digit-string

    digit::= 0 | 1 | ... | 9

    This is the equivalent in PVS:

    Number::= $Digit_+$

    Digit::= 0 | ... | 9

  - Reals

    Real literal constants in RSL are:

    real_lit::= digit-string '.' digit-string

    In PVS the type real is defined as:

    `real: NONEMPTY_TYPE FROM number`

    so `real` is an uninterpreted type in PVS which means that there are no real literal constants in PVS as in RSL. This implies transforming a real literal constant $i.d$ (where i is the integer part of the real number and d its decimal part) in RSL into a corresponding PVS of the form: $i + d/10^{dn}$ where dn is the number of decimal digits.

- **Characters**

  A Character in RSL has the following syntax:

  char_lit ::= 'char_character'

  char_character ::= character | quote

  character ::= ascii_letter | digit | graphic | escape

  ascii_letter ::= *capital and lower case letters*

  digit ::= 0 .. 9

  quote ::= ”

  escape ::= *escape characters* | *octal constant* | *hexadecimal constant*

  There is no provision in PVS for literals of type `char` as they are in RSL. Characters are defined as a `DATATYPE` in the `prelude`. So to obtain a character literal in PVS, the constructor `char` for the `DATATYPE` has to be used:

  `char(`n`)`, where $0 \leq n \leq 255$.

- **Texts**

  A **Text** in RSL has the following syntax:

  text_lit::= ” opt-text_character-string ”

  text_character ::= character | prime

In PVS this is:

String::= " *ASCII-character** "

RSL **Text** can be translated as PVS *String* that is defined in the `prelude` as a finite sequence of characters.

6.4.2 Names

In RSL when a value_expr is a name it must represent a value or a variable. A name in both RSL and PVS is an identifier or an operator.

The equivalent of an RSL value is a PVS constant. So when a name in RSL represents a value, in PVS the name represents a constant.

Variables on the other hand are not part of the transformation since they are used in RSL in imperative style specifications.

6.4.3 Prenames

Prenames are not accepted since they are used in RSL for variables in imperative style specifications.

6.4.4 Basic Expressions

Basic Expressions in RSL are **chaos**, **skip**, **stop**, **swap**. They are not accepted.

6.4.5 Product Expressions

Product value expressions in RSL are:

(value_expr-list2)

They are equivalent to PVS tuple expressions which have the form:

($Expr_{++}$)

6.4.6 Set Expressions

There are three kinds of set expressions in RSL: Ranged Set Expressions, Enumerated Set Expressions and Comprehended Set Expressions.

The transformation for each of them is given below.

Ranged Set Expressions

Ranged Set Expressions in RSL have the effect of constructing a set of integers made up of those integers in between a lower and upper bound.

RSL Construction 6.3
ranged_set_expr ::=
{ *readonly_integer*-value_expr$_1$ .. *readonly_integer*-value_expr$_2$ }
□

Since there is no equivalent in PVS, we can define the following function in PVS to help make the transformation:

PVS Construction 6.5
```
ranged_set(x: int, y: int) :
  setof[int] = LAMBDA (z: int) : x ≤ z AND z ≤ y
```
□

Then we can transform the RSL in Construction 6.3 into PVS as:

PVS Construction 6.6
`ranged_set(`*Expr*$_1$, *Expr*$_2$`)`
□

Enumerated Set Expressions

An Enumerated Set Expression in RSL is just a way to represent a set of explicitly specified values:

RSL Construction 6.4
enumerated_set_expr ::= { *readonly-opt*-value_expr$_1$, ..., *readonly-opt*-value_expr$_n$ }
□

In PVS there is not an in-built set constructor. An aproximate PVS construct would be:

(: *Expr*$_{**}$ ',' :)

But this implies using the automatic coercion from lists to sets since `(: :)` are the built-in lists constructors which will raise TCCs.

A more simpler construction would be using the set `add` function, and `emptyset` provided in the `prelude`. So {} in RSL can be translated as `emptyset` in PVS, and RSL Construction 6.4, then can be translated as:

PVS Construction 6.7
`add(`*Expr*$_n$`, ..., add(` *Expr*$_1$`, emptyset) ...)`

□

Comprehended Set Expressions

In RSL a Comprehended Set Expressions is a way of constructing a set by evaluating a value expression:

comprehended_set_expr ::= { *readonly*-value_expr | set_limitation }
set_limitation ::= typing-list opt-restriction
restriction ::= • *readonly_logical*-value_expr

In PVS this is the nearest equivalent construction:

{ *SetBindings* | *Expr* }

But this a more limited construct than the one in RSL and only covers some of the possibilities in RSL, namely those where the typing-list is single tying and the first value_expr is just the binding in the typing. So if we want to do a comlete transformation of this RSL construct we are obliged to use other additional PVS constructs.

Given any type T and U, a comprehended set expression in RSL can have the following general form:

RSL Construction 6.5
{ *f(x)* | *x: T* • *p(x)* }
where:
f, has type $T \xrightarrow{\sim} U$*;*
and
U-**infset**, *is the type of the value expression;*
and
p: $T \rightarrow$ **Bool**, *is a predicate on x*

□

then we can transform it into the following PVS general construct:

PVS Construction 6.8
```
{ u:  U | EXISTS (x:  T) : p(x) AND u = f(x)}
```

□

6.4.7 List Expressions

There are three kinds of list expressions in RSL: Ranged List Expressions, Enumerated List Expressions and Comprehended List Expressions.

The transformation $\mathcal{T}$ for each is given below.

Ranged List Expressions

In RSL, Ranged List Expressions have the effect of constructing a list of integers in a range bounded by a lower and upper bound.

RSL Construction 6.6
ranged_list_expr ::= ⟨ *integer*-value_expr$_1$ .. *integer*-value_expr$_2$ ⟩

□

Since there is no equivalent in PVS, we define a function called `ranged_list`, which uses a recursive function `ranged_list1`, to be able to make the transformation.

PVS Construction 6.9
```
ranged_list1(x: int) (y: { y : int | y ≥ x})
    : RECURSIVE list[int] =
      IF x = y
        THEN cons(x, null)
        ELSE cons(x, ranged_list1(x + 1)(y))
      ENDIF
MEASURE y - x
ranged_list(x: int, y: int) : list[ int ] =
      IF x > y
        THEN null
        ELSE ranged_list1(x)(y)
      ENDIF
```

□

Using the function defined in PVS Construction 6.9, we can now transform the RSL in RSL Construction 6.6 into the following PVS:

PVS Construction 6.10
`ranged_list(`*Expr*$_1$`,` *Expr*$_2$`)`

□

Enumerated List Expressions

An Enumerated List Expression in RSL is just a way to represent a list of explicitly specified values:

enumerated_list_expr ::= ⟨ opt-value_expr-list ⟩

PVS has an equivalent construct:

(: *Expr*$_{**}$ ',' :)

Comprehended List Expressions

In RSL a Comprehended List Expression is a way of constructing a list by evaluating a value expression:

comprehended_list_expr ::= ⟨ value_expr | list_limitation ⟩
list_limitation ::= binding **in** *readonly_list*-value_expr opt-restriction
restriction ::= • *readonly_logical*-value_expr

So given any two types T and U, a finite comprehended list expression is:

RSL Construction 6.7
⟨ *f(x)* | *x* **in** *L* • *p(x)* ⟩
where:
*L: T**
and
f: T → *U*
and
*U**, *is the type of the value expression*
and
p: T → **Bool**; *is a predicate on x and is the opt*-restriction

□

Considering that finite lists are predefined as a `DATATYPE` in PVS, if we define the following function in PVS:

PVS Construction 6.11

```
comp_list(l:list[ T], p:[ T → bool], f:[T → U]):  RECURSIVE list[U] =
   CASES l of
     null:  null,
     cons(h, t):
       IF p(h)
         THEN cons(f(h), comp_list(t, p, f))
         ELSE comp_list(t, p, f) ENDIF
   ENDCASES
MEASURE length(l)
```

□

then we can define the transformation $\mathcal{T}$ for the RSL Construction in 6.7 as producing the following PVS expression:

```
comp_list(L, (LAMBDA (x:  T) : p(x)), (LAMBDA (x:  T) : f(x)))
```

But we can instead employ a shorter transformation that uses the predefined PVS `map` function that generates the image of a sequence under a function. So in this case the transformation $\mathcal{T}$ for the RSL in Construction 6.7 will produce the following expression:

PVS Construction 6.12

```
map(LAMBDA (x:  T) : f(x), filter(L, LAMBDA (x:  T) : p(x)))
```

□

The function in PVS Construction 6.12 is better since is not only shorter but can exploit the existing `prelude` theorems about `map` and `filter`.

6.4.8 Map Expressions

There are two kinds of map expressions in RSL: Enumerated Map Expressions and Comprehended Map Expressions.

map_expr ::=
    enumerated_map_expr |
    comprehended_map_expr

Both are discussed below.

Enumerated Map Expressions

Enumerated Map Expressions have the following syntactic form:

[opt-value_expr_pair-list]
value_expr_pair ::= *readonly*-value_expr ↦ *readonly*-value_expr

And this is their general form:

RSL Construction 6.8

[$\mathsf{value\_expr}_1 \mapsto \mathsf{value\_expr}'_1, \ldots, \mathsf{value\_expr}_n \mapsto \mathsf{value\_expr}'_n$]
for $n \geq 0$

□

An Enumerated Map Expression is a way of given a map by explicitly enumerating its associations. An Enumerated Map Expression could return a non-determinstic map. This is an example of a non-deterministic Enumerated Map Expression in RSL:

[$1 \mapsto 2, 1 \mapsto 3$]

But this generates a confidence condition $1 \neq 1$, which can not be proved, so although it is accepted it is not *reliable*.

So having the following in our PVS `Map` theory:

PVS Construction 6.13

```
add_in_map(m:  map, d:  map_dom, r:  map_rng) :  map =
     m WITH [(d) := mk_rng(r)]
empty_map:  map = LAMBDA (d:  map_dom):  nil
```

□

the Enumerated Map Expression could have the general form:

PVS Construction 6.14

$\texttt{add\_in\_map}_1\texttt{(...add\_in\_map}_n\texttt{(empty\_map, } Expr_n\texttt{, } Expr'_n\texttt{)...}Expr_1\texttt{, } Expr'_1\texttt{)}$

□

Comprehended Map Expressions

Comprehended Map Expressions have the general form:

RSL Construction 6.9

[$\mathsf{value\_expr}_1 \mapsto \mathsf{value\_expr}_2$ | binding: type_expr • $\mathsf{value\_expr}_3$]

□

A Comprehended Map Expression allows the implicit definition of a map by giving a predicate —$\mathsf{value\_expr}_3$ in RSL Construction 6.9— that defines the associations.

We can transform the RSL into the following PVS lambda expression:

PVS Construction 6.15

```
LAMBDA (LambdaBinding:  TypeExpr_MapDom):
   IF EXISTS (Binding_RSL:  TypeExpr_RSL):
            Expr3 AND LambdaBinding = Expr1
      THEN mk_rng(Expr2) ELSE nil ENDIF
```

where:

LambdaBinding $\neq$ $Binding_{RSL}$
and not free in $TypeExpr_{RSL}$, $Expr_1$, $Expr_2$ *or* $Expr_3$.

□

6.4.9 Function Expressions

Function Expressions in RSL are of the form:

λ binding: type_expr • value_expr

These function expressions evaluate to a function. They are considered lambda abstractions representing a function of type:

$\mathsf{type\_expr} \xrightarrow{\sim} \mathrm{T}$

or

$\mathsf{type\_expr} \rightarrow \mathrm{T}$

if value_expr is convergent for all assigments of the binding to values in type_expr, where T is the type of value_expr.

In PVS there are lambda expressions that denote unnamed functions and they have the form:

`LAMBDA` *IdOps*`:` *TypeExpr*`:` *Expr*

and they also represent a function of type:

TypeExpr`->` `T`

where `T` is the type of *Expr*.

So provided we have value_expr$_{RSL} \equiv_{\mathcal{T}}$ *Expr*$_{PVS}$ we can transform an RSL Function Expression into a PVS Lambda Expression.

RSL Construction 6.10
value *f :* **Real** → **Real**
axiom $f \equiv \lambda\ x$: **Real** • **if** $x = 0.0$ **then** *0.0* **else** *1.0/x* **end**

□

This is the equivalent construct in PVS:

PVS Construction 6.16
```
f:  [real -> real]
f:  AXIOM f = LAMBDA (x:  real):
        IF x = 0 THEN 0 ELSE 1/x ENDIF
```

□

6.4.10 Application Expressions

Application expressions in RSL are obtained by applying a list, map, or function expression to an actual parameter.

So in RSL this is:

application_expr ::=
list_or_map_or_function-value_expr
actual_function_parameter-string
actual_function_parameter ::= (opt-value_expr-list)

In PVS an application expression is:

Expr Arguments
Arguments::= (*Expr*$_{++}$ ',')

However we will have to distinguish in PVS between the three applications since only function applications have the same format.

Map Application

Map application in PVS is obtained by applying the map function, but since the range of the map function is a `DATATYPE`, we have to use the corresponding `DATATYPE` *recognizer* (the equivalent in PVS of a RSL *observer*), which in this case is called `rng_part`. So given a map application in RSL as was shown above, in PVS it would take the following form:

`rng_part(`*map* `(`*map domain instance*`))`

List Application

List application in RSL is equivalent to list indexing:

RSL Construction 6.11
value_expr$_1$*(*value_expr$_2$*)*
where:
value_expr$_1$, *is a list expression;*
value_expr$_2$ *is an integer expression and*
$0 <$ value_expr$_2 \leq$ **len***(*value_expr$_1$*)*

□

This is similar in PVS to an application of the function `nth` defined in PVS `prelude theory` for lists as:

PVS Construction 6.17
```
nth(l, (n:below[length(l)])):  RECURSIVE T
  IF n = 0 THEN car(l) ELSE nth(cdr(l), n-1) ENDIF
MEASURE length(l)
```
where `T` *is the base type of the list.*

□

But as we can see in RSL Construction 6.11, in RSL indexing starts from 1 while in PVS — as we can see in PVS Construction 6.17 starts from 0, so the translation will have to transform value_expr$_2$ into (*Expr*$_2$ - 1).

6.4.11 Quantified Expressions

RSL has three quantifiers: ∀, ∃ and ∃!. They all serve to create quantified expressions of the form:

RSL Construction 6.12
quantifier typing$_1$, ..., typing$_n$ • value_expr
for $n \geq 1$

□

and where the typings define a set of models and where value_expr is a logical expression that evaluates to **true** or **false**.

Since quantified expressions in RSL are convergent, the meaning of the different quantified expressions in RSL are as can be expected:

- with ∀ the value returned is **true** if and only if the value_expr holds for all the models.
- with ∃ the value returned is **true** if and only if the value_expr holds for at least one of the models.
- with ∃! the value returned is **true** if and only if the value_expr holds for one and only one of the models.

PVS has predefined only two quantifiers: `FORALL` and `EXISTS`. They both are used in quantified expressions that the same form and the same meaning than RSL's ∀ and ∃:

PVS Construction 6.18
BindingOp LambdaBindings : Expr

□

where *BindingOp* can be `FORALL` or `EXISTS`, and *LambdaBindings* are equivalent to RSL's typing.

PVS defines in the prelude `exists1`, as a function, which can be used to represent the ∃! of RSL:

PVS Construction 6.19
```
x, y:  VAR T
p, q:  VAR pred[T]
unique?(p):  bool = FORALL x, y:  p(x) AND p(y) IMPLIES x = y
exists1(p):  bool = (EXISTS x:  p(x)) AND unique?(p)
```

□

So having in RSL:

RSL Construction 6.13
∃! *x: T* • *p*

□

this can be in PVS:

PVS Construction 6.20
```
exists1(LAMBDA(x:T): p
```

□

6.4.12 Equivalence Expressions

Equivalence Expressions in RSL are:

equivalence_expr ::= value_expr ≡ value_expr opt-pre_condition
pre_condition ::= **pre** *readonly_logical*-value_expr

With the opt-pre_condition present an Equivalence Expression is short for:

RSL Construction 6.14
*(*value_expr$_3$ ≡ **true***)* ⇒ value_expr$_1$ ≡ value_expr$_2$

□

Equivalence Expressions always evaluate to **true** or **false**, even when both value_exprs are non-deterministic. If value_expr$_1$ and value_expr$_2$ are convergent applicative expressions then equality (=) and equivalence (≡) mean the same.

Since there are no Equivalence Expressions in PVS, and we are accepting only RSL convergent applicative expressions (see Section 2.1, page 5) we can use PVS equality to transform Equivalence Expressions.

So the corresponding PVS for the RSL in Construction 6.14 is:

PVS Construction 6.21
Expr$_3$ `IMPLIES` *(Expr*$_1$ = *Expr*$_2$*)*

□

When the opt-pre_condition in RSL is not present the transformation should not produce the implication, it will be just:

PVS Construction 6.22
Expr$_1$ = *Expr*$_2$

□

6.4.13 Post Expressions

In RSL a Post Expression has the following syntax:

post_expr ::= value_expr post_condition opt-pre_condition
post_condition ::= opt-result_naming **post** *readonly_logical*-value_expr
result_naming ::= **as** binding

So in general we can say that a Post Expression in RSL is:

RSL Construction 6.15
value_expr$_1$ **as** binding
post value_expr$_2$
pre value_expr$_3$

□

A Post Expression when the opt-pre_condition is present is short for:

RSL Construction 6.16
(value_expr$_3$ ≡ **true**) ⇒ value_expr$_1$ **as** binding **post** value_expr$_2$

□

Where value_expr$_2$ is of type **Bool**, which is also the type of the Post Expression itself. A Post Expression is always defined and deterministic. The hooked variables that can appear in the binding apply to imperative RSL and are not accepted in the transformation.

The value of the Post Expression is **true** if and only if:

- after value_expr$_1$ is evaluated in the current state and it is defined and deterministic.
- value_expr$_2$ ≡ **true** when evaluated in the post-state and in the scope of the binding.

Pre and post-states are only of interest in imperative RSL and are not accepted in the transformation.

So given T as the type of value_expr$_1$ in RSL Construction 6.15, we can transform the RSL in 6.15 as the following PVS:

PVS Construction 6.23
Expr$_3$ `IMPLIES`
`(EXISTS (`*Binding*`: T):` *Binding* `=` *Expr*$_1$
`AND` *Expr*$_2$ `)`

□

When the opt-pre_condition is not present the transformation should not produce the implication, it will be just:

PVS Construction 6.24
`EXISTS (`*Binding*`: T):` *Binding* `=` *Expr*$_1$
`AND` *Expr*$_2$

□

6.4.14 Disambiguation Expressions

Disambiguation Expressions in RSL of the form value_expr : type_expr can be rendered in PVS by a Coercion Expression of the form: *Expr* :: *TypeExpr*, since the meaning of both is the same namely to resolve overloading of type-ambiguous expressions.

6.4.15 Bracketed Expressions

Bracketed Expressions in RSL which are of the form:

bracketed_expr : (value_expr)

have the equivalent PVS expression:

Expr::= (*Expr*)

6.4.16 Infix Expressions

There are in RSL three kinds of Infix Expressions. They are treated separately below.

Statement Infix Expressions

Statement Infix Expressions are used in the imperative sequential and concurrent style of specification, they are not accepted.

Axiom Infix Expressions

An Axiom Infix Expression in RSL is of the form:

RSL Construction 6.17
logical-value_expr$_1$ infix_connective *logical*-value_expr$_2$

□

Where value_expr$_1$ and value_expr$_2$ are boolean expresions.

This expression is equivalent in PVS to:

PVS Construction 6.25
Expr$_1$ *Binop* *Expr*$_2$

□

Value Infix Expressions

A Value Infix Expression in RSL is of the form:

RSL Construction 6.18
value_expr$_1$ infix_op value_expr$_2$

□

The maximal type of a Value Infix Expression in RSL is the result type part of the maximal type of the infix_op.

This expression is equivalent in PVS to:

PVS Construction 6.26
Expr$_1$ *Binop Expr*$_2$

□

where the infix_op has to be one of the accepted operators (see Section 7.2.1, page 74), otherwise the expression is not accepted.

6.4.17 Prefix Expressions

There are in RSL three kinds of Prefix Expressions. They are treated separately below.

Axiom Prefix Expressions

An Axiom Prefix Expression in RSL is of the form:

RSL Construction 6.19
prefix_connective *logical*-value_expr

□

Where value_expr is a boolean expresion and the prefix_connective is $\sim$.

This expression is equivalent in PVS to:

PVS Construction 6.27
Unaryop Expr

□

where *Unaryop* is `NOT`.

Universal Prefix Expressions

Universal Prefix Expressions are used in the imperative style of specification. They are also implicit in axioms, but can safely be omitted in applicative specifications. Therefore □value_expr in RSL is translated as *Expr* in PVS.

Value Prefix Expressions

A Value Prefix Expression in RSL is of the form:

RSL Construction 6.20
prefix_op value_expr

□

The maximal type of a Value Prefix Expression in RSL is the result type part of the maximal type of the prefix_op.

This expression is equivalent in PVS to:

PVS Construction 6.28
Unaryop Expr

□

where the prefix_op has to be one of the accepted operators (see section 7.2.2, page 84), otherwise the expression is not accepted.

6.4.18 Comprehended Expressions

Comprehended Expressions are used in the concurrent style of specification. They are not accepted.

6.4.19 Initialize Expressions

In RSL an Initialize Expression is used to give the initial value to a variable. These expressions are used in the imperative style of specification. They are not accepted.

6.4.20 Assignment Expressions

The effect of an Assignment Expression in RSL is to write a value to a variable. These expressions are used in the imperative style of specification. They are not accepted.

6.4.21 Input Expressions

Input Expressions offer to input from channels. They are used in the concurrent style of specification. They are not accepted.

6.4.22 Output Expressions

Output Expressions offer to output to channels. They are used in the concurrent style of specification. They are not accepted.

6.4.23 Structured Expressions

Structured Expressions in RSL are Local Expressions, Let Expressions If Expressions, Case Expressions, While Expressions, Until Expressions and For Expressions. Each is treated separately below.

6.4.24 Local Expressions

These expressions are not currently accepted.

6.4.25 Let Expressions

The syntax of Let Expression in RSL is:

RSL Construction 6.21

let_expr ::= **let** let_def-*list* **in** value_expr **end**
let_def ::= typing | explicit_let | implicit_let
explicit_let ::= let_binding = value_expr
implicit_let ::= single_typing restriction

□

And that of a Let Expression in PVS is:

PVS Construction 6.29

Expr::= `LET` $LetBinding_{++}$ ',' `IN` *Expr*
LetBinding::= { *LetBind* | $(LetBind_{++}$ ',') } = *Expr*
LetBind::= *IdOp Bindings* [: TypeExpr]*

□

Although the two constructs may look similar at first glance there is a crucial difference between them: PVS obliges one to have a definition of the *LetBind* while that is not the case in RSL for Implicit Let Expressions, which can be a non-deterministic expression.

So to avoid non-determinism, only RSL explicit_let expressions can be transformed into PVS `LET` expressions. RSL implicit_lets are not accepted.

The let_binding in RSL Construct 6.21 of an Explicit Let Expression has one of the three forms shown in RSL Construct 6.22.

RSL Construction 6.22

let_binding::=
- | binding
- | record_pattern::= *pure_value*-name(inner_pattern-*list*)
- | list_pattern::=
  - enumerated_list_pattern::= (*opt*-inner_pattern-*list*)
  - | concatenated_list_pattern::= enumerated_list_pattern ⌢ inner_pattern

□

Bindings in RSL are transformed into bindings in PVS (see Section 7.9, page 91), so an Explicit Let Expression with a binding transform to:

PVS Construction 6.30

`LET` *Binding* `=` *Expr*$_1$ `IN` *Expr*$_2$

□

Record and List Patterns have Inner Patterns, which have the following form:

RSL Construction 6.23

inner_pattern::=
- value_literal
- | id_or_op
- | wildcard_pattern
- | product_pattern
- | record_pattern
- | list_pattern
- | equality_pattern::= = *pure_value*-name

□

We can see that the Inner Patterns provide for recursion in patterns, we can transform list patterns by unfolding their definition and producing a list of bindings for them.

In the case of the Record Patterns we know that:

RSL Construction 6.24

let record_pattern = value_expr$_1$ **in** value_expr$_2$ **end** ≡
case value_expr$_1$ **of** record_pattern → value_expr$_2$ **end**

□

So we can transform a Let Expression with a Record Pattern into a PVS `CASES` expression with a single case.

6.4.26 If Expressions

If expressions in RSL are equivalent to `IF` expressions in PVS.

However in PVS `IF` expressions must have both branches present while in RSL **if** expressions without an **else** branch are permitted. This latter is equivalent to an **else** with a basic expression **skip** (that returns the unit value of type **Unit**) but this is a case of a value that is not accepted so only RSL **if** expressions with both branches are accepted in the transformation.

6.4.27 Case Expressions

Case Expressions in RSL have the following syntax:

case_expr ::= **case** value_expr **of** case_branch-list **end**
case_branch ::= pattern → value_expr

where pattern is one of the following:

RSL Construction 6.25
pattern ::=
value_literal
| *pure_value*-name
| wildcard_pattern
| product_pattern
| record_pattern
| list_pattern

□

Then we can say that the general form of a case expression is:

RSL Construction 6.26
case value_expr$_{test}$ **of**
pattern$_1$ → value_expr$_1$
⋮
pattern$_n$ → value_expr$_n$

□

PVS provides a `CASES` construct that is defined:

Expr::=
`CASES` *Expr* `OF`
Selection$_{++}$ ','
[`ELSE` *Expr*]
`ENDCASES`

and where

Selection::= *IdOp* | [`(` *IdOps* `)`] `:` *Expr*

and the *Selection* is defined only for records patterns of the form: $c(x_1, ..., x_n)$ and where c is an n-ary constructor (of a corresponding `DATATYPE`) and $x_1, ..., x_n$ is a list of distinct variables.

Actually PVS `CASES` construct provides a limited kind of pattern matching and only some of the more simple cases of the RSL **case** construct could be rendered with the PVS **cases** Expression and only when they correspond to an equivalent variant construct.

So to deal with the more general **case** construct of RSL, a transformation into a PVS `IF` expression is necessary.

A **case** expression of the form in the RSL Construction 6.26 can be transformed into a PVS `IF` expression of the form:

PVS Construction 6.31

```
IF matches(Expr_test, pattern_1)
  THEN LET defs(Expr_test, pattern_1) IN Expr_1
  ELSE
        ⋮
    IF matches(Expr_test, pattern_n-1)
      THEN LET defs(Expr_test, pattern_n-1) IN Expr_n-1
      ELSE LET defs(Expr_test, pattern_n) IN Expr_n
    ENDIF
      ⋮
ENDIF
```

□

where `matches(Expr, patt)` is the condition that `Expr` matches `patt` and `defs(Expr, patt)` are the resulting definitions: bindings of pattern variables to `Expr`. For example, suppose we have a list expression `l`. Then `matches(l,⟨⟩)` is `length(l) = 0`, and there are no definitions. For a pattern ⟨h⟩^t the matches condition will be `length(l) >= 1` and the definitions will be `h = car(l), t = cdr(l)`.

So we see that the case expression

case l **of**
  ⟨⟩ → value_expr$_1$,
  ⟨h⟩^t → value_expr$_2$
end

transforms to

```
IF len l = 0
  THEN Expr_1
  ELSE LET h = car(l), t = cdr(l) IN Expr_2
ENDIF
```

The `matches` condition is omitted from the last case, and so the reliability of the translation depends on checking the corresponding confidence condition that effectively says the cases are exhaustive. In this example the confidence condition is:

$$\textbf{len}\ l = 0 \vee \textbf{len}\ l \geq 1$$

which is obviously true.

6.4.28 While Expressions

RSL While Expressions are used in imperative RSL. They are not accepted.

6.4.29 Until Expressions

RSL Until Expressions are used in imperative RSL. They are not accepted.

6.4.30 For Expressions

RSL For Expressions are used in imperative RSL. They are not accepted.

7 Syntactic Issues

In producing a transformation from one language to another, obviously the syntax of both languages have to be taken into consideration, but the differences that can be found are of two kinds.

Since the syntax just expresses the semantics underlying it, the differences between the syntaxes can arise from two reasons. One is that the same semantics is expressed in a difference syntax, another is that differences in syntax reflects a difference in semantics.

The first case can have two sides: if the target language has such a construct then it becomes a question of finding the right syntax to represent it.

Sometimes, although the semantics is provided directly in the target language, it does so only partially, or only through a different approach. This is the most simple case.

When the difference in syntax reflects a difference in semantics then it can happen that the transformation is impossible if the target language does not provide the corresponding constructs for it. Otherwise it becomes a question of finding the semantic equivalent.

The particular syntactic issues are discussed in detail below.

7.1 Identifiers

Identifiers and operators are either an instance of a defining occurrence or an applied occurrence in RSL. The same is true for PVS.

Each RSL operator and its corresponding PVS one are treated separately below.

In general RSL and PVS —as do most other computer languages, being they programming languages or otherwise— have nearly the same kind of identifiers.

However RSL allows certain characters that are not normally used in this context.

Greek letters can be part of the chain of characters in identifiers in RSL.

The ASCII version of Greek letters are preceded by an opening single quotation mark followed by the English name of the letter. See the table in [6], pages 384 and 385.

Also strings of closing quotation marks (primes) are allowed in identifiers.

Since in PVS identifiers must start with a letter, one way to deal with RSL Greek letters is just

to transform the opening single quotation mark into a particular string of characters, starting with a letter, that is used for just this purpose.

A similar solution can be adopted for the string of primes in RSL identifiers.

To be consistent with other tools in the RSL set of tools —RSL to C++ translator, for example— we proposed to adopt the same convention. See Section 8, page 96 for more information about the particular strings that are used in this convention.

7.2 Operators

Some RSL built-in operators do not exist as such in PVS but are declared as functions in PVS `prelude` theory. This does not present a particular problem except that the presentation sometimes will have to change from an infix or prefix form to a form such as `f(x)`.

The same is valid when there are operators that are not in PVS and have to be generated ad hoc.

7.2.1 Infix Operators

- **Equality**
  - RSL
    $= : \mathrm{T} \times \mathrm{T} \rightarrow \mathbf{Bool}$
  - PVS
    `= :  [T, T -> bool]`
- **Inequality**
  - RSL
    $\neq : \mathrm{T} \times \mathrm{T} \rightarrow \mathbf{Bool}$
  - PVS
    `/= :  [T, T -> bool]`
- **Integer addition**
  - RSL $+ : \mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int}$
  - PVS
    `+ :  [int, int -> int]`
- **Real addition**

  - RSL
    $+ : \mathbf{Real} \times \mathbf{Real} \rightarrow \mathbf{Real}$
  - PVS
    `+ : [real, real -> real]` ‘

- **Integer subtraction**

  - RSL
    $- : \mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int}$
  - PVS
    `- : [int, int -> int]`

- **Real subtraction**

  - RSL
    $- : \mathbf{Real} \times \mathbf{Real} \rightarrow \mathbf{Real}$
  - PVS
    `- : [real, real -> real]`

- **Integer multiplication**

  - RSL
    $* : \mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int}$
  - PVS
    `* : [int, int -> int]`

- **Real multiplication**

  - RSL
    $* : \mathbf{Real} \times \mathbf{Real} \rightarrow \mathbf{Real}$
  - PVS
    `* : [real, real -> real]`

- **Integer exponentiation**

  - RSL
    $\uparrow : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$
    $a \uparrow b$ **pre**
    if $a = 0$ **then** $b > 0$ **else** $b \geq 0$
  - PVS
    An operator that could be used to transform the RSL operator is $\widehat{}$ which is defined in the `prelude theory` in terms of `expt` —which is the real exponentiation— to work for integers as:

```
^ (r:  real, i:{i:int | r /= 0 OR i > = 0}):  real =
  IF i >= 0 THEN expt(r, i) ELSE 1/expt(r, -i) ENDIF
```

It has an integer exponent and returns a real. It could be used since when the RSL pre condition holds (checked by the confidence condition) then the result will be an integer. We prefer to define a function:

```
rsl_expt(a:int, b:{b:  int | IF a = 0
                             THEN b > 0
                             ELSE b >= 0 ENDIF}
         ):  int = a ^ b
```

- **Real exponentiation**
  - RSL
    $\uparrow$: **Real** $\times$ **Real** $\xrightarrow{\sim}$ **Real**
    Pre-condition: Given a $\uparrow$ b; if b < 0.0 then a $\neq 0.0$. If b is not a whole number then a ≥ 0.0.
  - PVS
    The nearest equivalent operator that can be used to transform the RSL operator is `expt` which is defined in the `prelude theory` as follows:

```
r:  VAR real
m, n:  VAR nat
expt(r, n):  RECURSIVE real =
   IF n = 0 THEN 1
   ELSE r * expt(r, n-1)
   ENDIF
MEASURE n;
```

    But PVS function is defined for exponents of type `nat`, while in RSL exponents can be of type **real**, so exponentiation of reals other than with a natural exponent could not be transformed, and this will oblige to recognize when an RSL exponent is a natural number, which is a non-decidable problem. So this operation is not accepted.

- **Function Composition**
  - RSL
    $\circ$: $(T_2 \xrightarrow{\sim} a\ T_3) \times (T_1 \xrightarrow{\sim} a'\ T_2) \rightarrow (T_1 \xrightarrow{\sim} a''\ T_3)$
    where a" is: a $\cup$ a'
    $f_1 \circ f_2 \equiv \lambda x\ T_1 \bullet f_1(f_2(x))$
  - PVS

```
f1:  VAR [T1 -> T2]
f2:  VAR [T2 -> T3]
o(f1, f2)(x):  T3 = f2(f1(x))
```

- **Map Composition**
  - RSL
    $\circ : (T_2 \xrightarrow{m} T_3) \times (T_1 \xrightarrow{m} T_2) \rightarrow (T_1 \xrightarrow{m} T_3)$
    Which is defined as a comprehended map expression is:
    $m_1 \circ m_2 \equiv [\ x \mapsto m_1(m_2(x)) \mid x: T_1 \bullet x \in \textbf{dom}\ m_2 \wedge m_2(x) \in \textbf{dom}\ m_1\ ]$
  - PVS
    Since there is no map composition operator in PVS we define this operator as a function in the `Map THEORY` as follows:

```
M1:  TYPE = map[T2, T3]
M2:  TYPE = map[T1, T2]
M3:  TYPE = map[T1, T3]

map_o(m1:  M1, m2:  M2):  M3 =
  LAMBDA (xl:  T1):
       IF   member(x, dom(m2)) AND
            member(rng_part(m2(x)), dom(m1))
         THEN m1(rng_part(m2(x)))
         ELSE nil
       ENDIF
```

- **Integer division**
  - RSL
    $/ : \textbf{Int} \times \textbf{Int} \xrightarrow{\sim} \textbf{Int}$
    a / b **pre** b $\neq 0$
  - PVS
    Division is defined for reals in PVS. Integer is a subtype of real, and as has already been said there is no separate operators for integers and reals.
    We can define a function in PVS to do this:

```
m, n:  VAR real
a:  VAR int
b:  VAR nzint
rsl_int_div(a, b) :  int = sgn(a) * sgn(b) * ndiv(abs(a),abs(b))
```

    and where `ndiv`, `sgn` and `abs` are defined in PVS `prelude` as:

```
ndiv(x,b) :  {q:  int | x = b * q + rem(b)(x)}
sgn(m):  int = IF m >= 0 THEN 1 ELSE -1 ENDIF
abs(m):  {n:  nonneg_real | n >= m} = IF m < 0 THEN -m ELSE m ENDIF
```

- **Real division**
  - RSL
    / : **Real** × **Real** $\xrightarrow{\sim}$ **Real**
    a / b **pre** b ≠ 0.0
  - PVS
    ```
    nonzero_real:NONEMPTY_TYPE = {r:real | r /= 0} CONTAINING 1
    nzreal:  NONEMPTY_TYPE = nonzero_real
    /:  [real, nzreal -> real]
    ```

- **Map restriction to**
  - RSL
    / : (D $\xrightarrow[m]{}$ R) × S → (D $\xrightarrow[m]{}$ R)
    m \ s = [d ↦ m(d) | d: D • d ∈ **dom** m ∧ d ∈ s
  - PVS
    This operator is defined in the `Map THEORY` as:
    ```
    restriction_to(m:  map, s:  set[map_dom]):  map =
     LAMBDA (d:  map_dom) :
      IF member(d, dom(m)) AND member(d, s)
       THEN m(d)
       ELSE nil
      ENDIF
    ```

- **Integer remainder**
  - RSL
    \ : **Int** × **Int** $\xrightarrow{\sim}$ **Int**
    a \ b **pre** b ≠ 0
    ⇒ a = (a / b) * b + (a \ b)
    In RSL the result of the remainder operation is an integer. It takes the sign of the first argument.
  - PVS
    In PVS the remainder operation has a different signature than the one in RSL, it always returns a natural. So to keep consistency this operation will have to be redefined to include RSL definition, changing the sign of the operation as needed.
    ```
    m, n :  VAR real
    x :  VAR int
    b:  VAR posnat
    ```
    This is the new remainder operation to keep consistent with RSL's:
    ```
    rsl_int_rem(a, b) :  int = sgn(a) * rem(abs(b))(abs(a))
    ```

and where `rem`, `mod`, `sgn` and `abs` are defined in PVS `prelude` as:

```
rem(b)(x):  {r:  mod(b) | EXISTS q:  x = b * q + r}
mod(b) :  NONEMPTY_TYPE = { i:  nat | i < b}
sgn(m):  int = IF m >= 0 THEN 1 ELSE -1 ENDIF
abs(m):  n:  nonneg_real | n >= m = IF m < 0 THEN -m ELSE m ENDIF
```

- **Set difference**
  - RSL
    $\setminus$: T-**infset** $\times$ T-**infset** -¿ T-**infset**
  - PVS
    `difference(a, b):  set = {x | member(x, a) AND NOT member(x, b)}`
- **Map restriction by**
  - RSL
    $\setminus : (D \xrightarrow[m]{} R) \times S \to (D \xrightarrow[m]{} R)$
    $m \setminus s = [d \mapsto m(d) \mid d: D \bullet d \in \mathbf{dom}\ m \wedge d \notin s$
  - PVS
    This is defined in the `Map THEORY` as follows:

```
restriction_by(m:  map, s:  set[map_dom]):  map =
 LAMBDA (d:  map_dom) :
  IF member(d, dom(m)) AND NOT member(d, s)
   THEN m(d)
   ELSE nil
  ENDIF
```

- **Integer greater than**
  - RSL
    $>$: **Int** $\times$ **Int** $\to$ **Bool**
  - PVS

```
x, y:  VAR real
>(x, y):  bool = y < x
```

- **Real greater than**
  - RSL
    $>$: **Real** $\times$ **Real** $\to$ **Bool**
  - PVS

```
x, y:  VAR real
>(x, y):  bool = y < x
```

- **Integer less than**
  - RSL
    $<$: **Int** $\times$ **Int** $\rightarrow$ **Bool**
  - PVS
    ```
    x, y:  VAR real
    <(x, y):  bool
    ```
- **Real less than**
  - RSL
    $<$: **Real** $\times$ **Real** $\rightarrow$ **Bool**
  - PVS
    ```
    x, y:  VAR real
    <(x, y):  bool
    ```
- **Integer greater than or equal to**
  - RSL
    $\geq$: **Int** $\times$ **Int** $\rightarrow$ **Bool**
  - PVS
    ```
    x, y:  VAR real
    >=(x, y):  bool = y <= x
    ```
- **Real greater than or equal to**
  - RSL
    $\geq$: **Real** $\times$ **Real** $\rightarrow$ **Bool**
  - PVS
    ```
    x, y:  VAR real
    >=(x, y):  bool = y <= x
    ```
- **Integer less than or equal to**
  - RSL
    $\leq$: **Int** $\times$ **Int** $\rightarrow$ **Bool**
  - PVS
    ```
    x, y:  VAR real
    <=(x, y):  bool = x < y OR x = y
    ```
- **Real less than or equal to**
  - RSL
    $\leq$: **Real** $\times$ **Real** $\rightarrow$ **Bool**

  - PVS

    ```
    x, y:  VAR real
    <=(x, y):  bool = x < y OR x = y
    ```

- **Set superset (proper)**

  - RSL

    $\supset$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  - PVS

    This operator is not defined in PVS. One way of solving the problem is to use `strict_subset?` operator, that is defined in the `prelude theory` as a function, with the parameters reversed instead. A better solution is to define a new operator in PVS:

    ```
    a, b:  VAR setof[T]
    strict_supset?(a, b):  bool = strict_subset?(b, a)
    ```

- **Set subset (proper)**

  - RSL

    $\subset$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  - PVS

    ```
    set:  TYPE = setof[T]
    x, y:  VAR T
    a, b, c:  VAR set
    strict_subset?(a, b):  bool = subset?(a, b) & a /= b
    ```

- **Set superset**

  - RSL

    $\supseteq$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  - PVS

    This operator is not defined in PVS. One way of solving the problem is to use `subset?` operator, that is defined in the `prelude theory` as a function, with the parameters reversed instead. A better solution is to define a new operator in PVS:

    ```
    a, b:  VAR setof[T]
    supset?(a, b):  bool = subset?(b, a)
    ```

- **Set subset**

  - RSL

    $\subseteq$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  - PVS

```
set:  TYPE = setof[T]
x, y:  VAR T
a, b, c:  VAR set
subset?(a, b):  bool = (FORALL x:  member(x, a) => member(x, b))
```

- **Set membership**

  - RSL

    $\in$: T $\times$ T-**infset** $\rightarrow$ **Bool**

  - PVS

```
set:  TYPE = setof[T]
x, y:  VAR T
a, b, c:  VAR set
member(x, a):  bool = a(x)
```

- **Set membership (negation)**

  - RSL

    $\notin$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  - PVS

    This operator is not defined in PVS. One way of solving the problem is to negate the function `member` that is defined in the `prelude theory`. Another solution would be to define a new operator in PVS:

```
x:  VAR T
a:  VAR setof[T]
not_member(x, a):  bool = NOT member(x, a)
```

- **Set intersection**

  - RSL

    $\cap$: T $\times$ T-**infset** $\rightarrow$ T-**infset**

  - PVS

```
set:  TYPE = setof[T]
x, y:  VAR T
a, b, c:  VAR set
intersection(a, b):  set = {x | member(x, a) AND member(x, b)}
```

- **Set union**

  - RSL

    $\cup$: T-**infset** $\times$ T-**infset** $\rightarrow$ T-**infset**

  - PVS

```
set:  TYPE = setof[T]
x, y:  VAR T
a, b, c:  VAR set
union(a, b):  set = {x | member(x, a) OR member(x, b)}
```

- **Map union**

  - RSL

    $\cup : (\mathrm{D} \xrightarrow[m]{} \mathrm{R}) \times (\mathrm{D} \xrightarrow[m]{\sim} \mathrm{R}) \to (\mathrm{D} \xrightarrow[m]{} \mathrm{R})$

  - PVS

    This operator is defined in the `Map THEORY` as follows:

```
union:  [map, map -> map]
disjoint(m1, m2:  map) :  bool =
 FORALL (d:  map_dom) :
  NOT (member(d, dom(m1)) AND
   member(d, dom(m2)))
disjoint:  AXIOM FORALL (m1, m2:  map) :
 disjoint(m1, m2) IMPLIES
  union(m1, m2) = override(m1, m2)
```

    We have to use the `disjoint` axiom to define `union` in PVS to avoid having non-determinism.

- **List concatenation**

  - RSL

    $\frown : \mathrm{T}^{\omega} \times \mathrm{T}^{\omega} \xrightarrow{\sim} \mathrm{T}^{\omega}$

    pre first list must be finite.

  - PVS

    In PVS the `append` operation, that is the equivalent to the RSL $\frown$ operation on lists, is defined recursively for finite lists.

```
list [T: TYPE]: DATATYPE
BEGIN
 null:  null?
 cons (car:  T, cdr:list):cons?
END list
l, l1, l2, l3:  VAR list[T]
x:  VAR T
append(l1, l2):  RECURSIVE list[T] =
 CASES l1 OF
  null:  l2,
```

```
 cons(x, y):  cons(x, append(y, l2))
ENDCASES
MEASURE length(l1)
```

- **Map override**
  - RSL
    $\dagger : (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_2) \times (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_2) \to (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_2)$
  - PVS
    This operator is defined in the `Map THEORY` as follows:

```
override(m1, m2:  map):  map =
 LAMBDA (d:  map_dom) :
  IF member(d, dom(m2))
   THEN m2(d)
   ELSE IF member(d, dom(m1))
         THEN m1(d)
         ELSE nil
        ENDIF
  ENDIF
```

    A special case of the map override in PVS is using the PVS operator `:=`, so having:
    `m :  map[D, R]`
    `d:  D , r:  R`
    then we can use:
    `m WITH [d := mk_rng(r)]`
    as translation of m † [d ↦ r]

7.2.2 Prefix Operators

- **Integer absolute value**
  - RSL
    abs : **Int** → **Int**
  - PVS

```
abs(m):  {n:  nonneg_real | n >= m}
= IF m < 0 THEN -m ELSE m ENDIF
```

- **Real absolute value**
  - RSL
    abs : **Real** → **Real**

  - PVS
    ```
    abs(m):  {n:  nonneg_real | n >= m}
    = IF m < 0 THEN -m ELSE m ENDIF
    ```

- **Real to integer conversion**
  - RSL
    int : **Real** → **Int**
  - PVS
    There is no equivalent in PVS since both reals and integers are subtypes of the type `numbers` and conversions are done automatically. So **int** is just ignored in the translation.

- **Integer to real conversion**
  - RSL
    real : **Int** → **Real**
  - PVS
    There is no equivalent in PVS since both reals and integers are subtypes of the type numbers and conversions are done automatically. So **real** is just ignored in the translation.

- **Set cardinality**
  - RSL
    card : T-**infset** $\xrightarrow{\sim}$ **Int**
    However the effect of applying **card** to an infinite set is to diverge.
  - PVS
    There is no cardinality operator in PVS. A possible solution would be to define a function `card` as:
    ```
    ST: finite_set[T]
    card:  [ST -> nat]
    card_ax:  AXIOM FORALL (s:  ST):
       card(s) =
         IF nonempty?(s)
            THEN card(remove(choose(s), s)) + 1
            ELSE 0
         ENDIF
    ```
    This is defined in PVS only for finite sets.

- **List length**
  - RSL
    len : T^{ω} $\xrightarrow{\sim}$ **Int**

– PVS

The length operator is defined in PVS `prelude` as follows:

```
l:  VAR list[T]
x:  VAR T
length(l):  RECURSIVE nat =
  CASES l OF
   null:  0,
   cons(x, y):  length(y) + 1
  ENDCASES
  MEASURE reduce_nat(0, (LAMBDA (x:  T), (n:  nat):  n + 1))
```

- **List indexes**

  – RSL

  inds : $T^{\omega} \rightarrow$ **Int-infset**

  – PVS

  There is no indices operator in PVS. A possible solution would be to define a function `inds` as:

```
LT: TYPE = list[T]
inds:  [LT -> setof[nat]]
inds_ax:  AXIOM FORALL (l:  LT):
   inds(l) =
     CASES l OF
        cons(h, t):  add(length(l) , inds(t)),
        null:  emptyset
     ENDCASES
```

  A more elegant and shorter solution would be:

```
inds:  (l:  LT): setof[nat] =
LAMBDA: (n:  nat):  n > 0 AND n <= length(l)
```

  The second solution is adopted.

- **List elements**

  – RSL

  elems : $T^{\omega} \rightarrow$ T**-infset**

  – PVS

  An operator equivalent to **elems** is defined in PVS `prelude` as follows:

```
list2set(l) :  RECURSIVE set[T] =
  CASES l OF
    null:  emptyset[T],
    cons(x, y):  add(x, list2set(y))
  ENDCASES
MEASURE length
```

- **List head**
  - RSL
    hd : $T^{\omega} \xrightarrow{\sim} T$
    pre list non-empty.
  - PVS
    Lists are defined in PVS `prelude theory` as a `DATATYPE` with one constructor being: `cons (car: T, cdr:list):cons?`, so `car` can be used as the operator equivalent to **hd**.

- **List tail**
  - RSL
    tl : $T^{\omega} \xrightarrow{\sim} T^{\omega}$
    pre list non-empty.
  - PVS
    Lists are defined in PVS `prelude theory` as a `DATATYPE` with one constructor being: `cons (car: T, cdr:list):cons?`, so `cdr` can be used as the operator equivalent to **tl**.

- **Map domain**
  - RSL
    dom : $(T_1 \xrightarrow[m]{} T_2) \rightarrow$ T_1-**infset**
  - PVS
    This operator is defined in the `Map THEORY` as follows:

```
dom(m: map) : set[map_dom] =
  (LAMBDA(d: map_dom) : nonnil?(m(d)))
```

    So given the following RSL:
    x ∈ **dom** m
    It can be transformed into the corresponding PVS construct thus:
    `member(x, dom(m))`

- **Map range**
  - RSL
    rng : $(T_1 \xrightarrow[m]{} T_2) \rightarrow$ T_2-**infset**
  - PVS
    This operator is defined in the `Map THEORY` as follows:

```
rng(m: map): set[map_rng]
  (LAMBDA (r: map_rng) :
   EXISTS (d: map_dom) :
   nonnil?(m(d)) AND r = rng_part(m(d)))
```

    So given the following RSL:

x ∈ **rng** m

It can be transformed into the corresponding PVS construct thus:

```
member(x, rng(m))
```

7.3 Connectives

7.3.1 Infix Connectives

Boolean infix connectives are defined in RSL in terms of **if** expressions. PVS has similar `POSTULATES` which define the connectives in the prelude. This definitions are equivalent as we can see below:

- **AND**
  - **RSL** $value\_expr_1 \wedge value\_expr_2$

    if value_expr_1 **then** value_expr_2 **else false end**
  - **PVS** `A AND B`

    ```
    and_def:  POSTULATE (A and B) = IF A THEN B ELSE false ENDIF
    ```
- **OR**
  - **RSL** $value\_expr_1 \vee value\_expr_2$

    if value_expr_1 **then true else** value_expr_2 **end**
  - **PVS** `A OR B`

    ```
    or_def:  POSTULATE (A or B) = IF A THEN true ELSE B ENDIF
    ```
- **IMPLIES**
  - **RSL** $value\_expr_1 \Rightarrow value\_expr_2$

    if value_expr_1 **then** value_expr_2 **else true end**
  - **PVS** `A IMPLIES B`

    ```
    implies_def:  POSTULATE (A implies B) = IF A THEN B ELSE true ENDIF
    ```

7.3.2 Prefix Connectives

RSL and PVS have the same prefix connective: $\sim$ in RSL, `NOT` in PVS, and they are both define the same. In RSL:

$\sim$value_expr

is short for:

if value_expr **then false else true end**

and in PVS:

```
not_def:  POSTULATE (not A) = IF A THEN false ELSE true ENDIF
```

The definitions are equivalent.

7.4 Infix Combinators

No equivalent in PVS transformation since they are meant, in RSL, for concurrent specifications. They are not accepted.

7.5 Overloading

Overloading of identifiers is permitted in both RSL and PVS. However RSL has a stronger name resolution methodology than PVS when using name qualification from modules. This is considered in this report in the section dealing with modules.

In RSL an identifier or operator is overloaded at a certain point if there are several definitions of that identifier or operator which are visible at that point. Only value identifiers and operators are allowed to be overloaded.

PVS allows operator overloading. Declaration identifiers are classified according to *kind*, and these are: *type*, *prop*, *expr* and *theory*. New names have to be unique within a *kind*. Any overloading will be resolved at that level.

In RSL for a given construct the *legal interpretations* of the applied occurrences of identifiers and operators are found in the following way:

- if the construct has no subconstructs then consider all combinations of possible interpretations of the identifiers and operators. Otherwise consider all possible combinations of interpretations which are legal for subconstructs of the construct.
- then remove those combinations which do not satisfy the context conditions for the construct.
- finally, if the construct is a value expression (belongs to the syntactic category value_expr) then remove those combinations for which the construct has indistinguishable maximal type.

We have indicated above that PVS declaration identifiers are classified according to *kind* and that the overloading is resolved at the level of the *kind*.

Since in RSL the maximal type plays an important role in resolving overloading conflicts while in PVS is the concept of *kind* that is used for resolving conflicts some problems may arise in case that there are some overloadings accepted in RSL that can not be resolved in PVS. However PVS seems to resolve overloading inside a *kind* using maximal types.

For example the following is accepted by PVS Type Checker:

```
x : int
x : char
```

Both declarations of the constant `x` are of the same *kind* but they are distinguished by the different *maximal* type.

7.6 Names

A name in RSL represents a scheme, theory, type, value, object, variable or channel. Except for the last three —object, variable, channel— identifiers in PVS represent theories, types and constants which are the equivalent to the scheme, theory, type and value of RSL.

A name in RSL can be an identifier (id) or an operator (op).

A name (*Name*) in PVS can also be an identifier (*Id*) or an operator (*Opsym*).

7.6.1 Name Qualification

A name in RSL can have a qualification and a name in PVS can also have a qualification. In both cases this serves the purpose of qualifying names that are in different modules.

However RSL accepts nested qualifiers while PVS does not.

name ::= qualified_id ::= — qualified_op
qualified_id ::= opt-qualification id
qualified_op ::= opt-qualification(op)
qualification ::= *element*-object_expr

and

object_expr ::= name — ...

But this is directly connected with the way we treat modules (section 4, page 27).

7.7 Define Before Use

In RSL the order of declarations inside a module is immaterial. In PVS declarations are ordered within a theory. So PVS does not allow that earlier declarations reference later ones. This creates the need of ordering the declarations done in RSL to respect the PVS principle of 'define before use'.

7.8 Other Issues

Comments are not carried over from the the RSL form into PVS. This is difficult since most of the time the meaning of comments are intimately related to the positions of language constructs and the transformation can not guarantee the same format from one to the other.

It is also hard to do in a tool that discards comments when parsing, so there are none in the abstract syntax tree.

7.9 Bindings and Typings

A typing in RSL is a generalized declarative construct to bound identifiers —a binding— to types. This corresponds to PVS construct *Bindings*, which is also a way to bound identifiers —*IdOp*— to types.

Typings and Bindings in RSL and PVS

RSL	PVS
typing ::= single_typing | multiple_typing single_typing ::= binding : type_expr multiple_typing::= binding-list2 : type_expr binding ::= id_or_op | product_binding	*Bindings* ::= (*Binding*++ ',') *Binding* ::= *TypeId* | { (*TypeIds*) } *TypedIds* ::= *IdOps* [: *TypeExpr*] [| *Expr*] *TypedId* ::= *IdOp* [: *TypeExpr*] [| *Expr*]

Typings and Bindings in RSL and PVS

RSL	PVS
product_binding ::= (binding-list2) id_or_op	 *IdOp*

Bindings and typings are used in the language every time an identifier has to be defined and a type has to be associated with it (typings).

RSL typing and binding are used throughout several constructs in the language. PVS has some other constructs —apart from *Bindings*— that are used in special cases. The tables below show some of the most important correspondences.

However from Table 1 we can see a crucial difference in both constructs: RSL typing and binding are defined in the language as recursive grammatical structures while PVS *Bindings* are not. We have to take into consideration the fact that in RSL there is the additional typing-list as a construct to be taken care of.

This recursion in RSL serves specially for defining products —which correspond to `tuples` in PVS. The way to handle `tuples` in PVS is using projections over a `tuple`. So the solution is transforming RSL typing and binding into PVS projections by generating a special identifier and then use PVS projection operator —`‘1, ‘2,...` or `PROJ_1, PROJ_2,...`, where the first format is preferred— with this identifier. In Table 2 we show the proposed correspondence between RSL typing-list construct and *Bindings* in PVS. The typing-list construct appears in several RSL constructs: comprehended set expressions, quantified expressions, etc., and we want to show how we propose to deal with it.

Typing list in RSL and Bindings in PVS

RSL	PVS
typing-list | typing ::= | single_typing ::= binding ::= | id_or_op : type_expr	*Bindings* ::= (*Binding*++ ’,’) (*Binding*) *Binding* ::= | *TypedId* ::= *IdOp* [: *TypeExpr*]
| product_binding ::= (binding-list2) : type_expr	Generate an IdOp and a list of projections from the binding-list2 [: *TypeExpr*]
| multiple_typing ::= binding-list2	*Bindings* ::= (*Binding*++ ’,’) Generate an *IdOp* and a list of projections from the binding-list2

Typing list in RSL and Bindings in PVS

RSL	PVS
: type_expr	Generate a tuple from [: *TypeExpr*]
| typing-list	*Bindings* ::= (*Binding*++ ',')

Table 3 shows the way we propose to deal with the transformation of the single_typing in a Subtype Expression in RSL into the *SetBindings* in a PVS subtype expression. Subtype Expressions are dealt with in Section 6.3.7, page 50. Here we show only how we deal with the corresponding typings. The other constructs are there just as indications of the general expression format.

Typings in Subtype expression in RSL and in PVS

RSL	PVS
subtype_expr ::= {| single_typing binding ::= | id_or_op : type_expr	*Subtype* ::= { *SetBindings* ::= | *SetBinding* ::= | *IdOp* [: *TypeExpr*]
| product_binding ::= (binding-list2) : type_expr value_expr |}	Generate an *IdOp* and a list of projections from the binding-list2. This is equivalent to: *SetBindings*::= *SetBinding*::= *IdOp* [: *TypeExpr*] *Expr* }

Table 4 shows the way we propose to deal with the transformation of the typings in a comprehended set expression in RSL into the *SetBindings* in an equivalent PVS expression. Comprehended Set Expressions are specially dealt with in Section 6.4.6, page 55. Here we show only how we deal with the corresponding typings. The other constructs are there just as indications of the general expression format.

Typings in Comprehended Set Expression in RSL and PVS

RSL	PVS
{value_expr$_1$ set_limitation ::= typing-list | typing ::=	{ *SetBindings* ::= *LambdaBindings* ::= *Bindings* ::= (*Binding*++ ',') (*Binding*)

Typings in Comprehended Set Expression in RSL and PVS

RSL	PVS
| single_typing ::= binding ::= | id_or_op	*Binding* ::= | *TypedId* ::= *IdOp*
| product_binding ::= (binding-list2) : type_expr	Generate an *IdOp* and a list of projections from the binding-list2 This is equivalent to: *SetBindings*::= *SetBinding*::= *IdOp* [: *TypeExpr*]
| multiple_typing ::= binding-list2 : type_expr	*Bindings* ::= (*Binding*++ ',') Generate an *IdOp* and a list of projections from the binding-list2 This is equivalent to: *SetBindings*::= *SetBinding*::= *IdOp* Generate a tuple from [: *TypeExpr*]
| typing-list	*Bindings* ::= (*Binding*++ ',')
restriction(value_expr$_2$) }	restriction(*Expr*$_2$) }

Table 5 shows the way we propose to deal with the transformation of the typings in a function expression in RSL into the *LambdaBindings* in an equivalent PVS expression. Function Expressions are specially dealt with in Section 6.4.9, page 59. Here we show only how we deal with the corresponding typings, the other constructs are there just as indications of the general expression format.

Typings in Function Expressions in RSL and PVS

RSL	PVS
function_expr ::= λ lambda_parameter ::= | lambda_typing ::= (typing-list) typing ::= | single_typing ::= binding ::= | id_or_op	BindingExpr ::= LAMBDA LambdaBinding ::= | Bindings ::= (Binding++ ',') (Binding) Binding ::= TypedId ::= IdOp
| product_binding ::= (binding-list2)	Generate an *IdOp* and a list of projections from the binding-list2.

Typings in Function Expressions in RSL and PVS

RSL	PVS
	This is equivalent to: *SetBindings*::= *SetBinding*::= *IdOp*
| multiple_typing ::= binding-list2 : type_expr	 [: TypeExpr]
typing-list	(Binding++ ',')
| single_typing ::= binding ::= | id_or_op : type_expr	| Binding ::= TypedId ::= IdOp [: TypeExpr]
value_expr	Expr

We have shown through a number of tables the transformation of Bindings and Typings in RSL to the corresponding PVS constructs. The main PVS construct employed is the projection operator —‘ or `PROJ_`— that allows the transformation of the recursive structure in RSL into a suitable PVS one, specially when we have to deal with RSL products and their transformation into PVS tuples.

8 Notes on the Tool

8.1 Introduction

The tool that implements the Transformation of RSL into PVS can be considered a Demonstrator of the principles used in the Transformation, namely the automatism provided by the tool is a empirical way to demonstrate —every time the tool is used to do a translation— that the translation is correct.

Actually the resulting PVS code generated from the input RSL code is ready to be submitted for a practical test of its soundness. Every transformation using the tool can be seen as a new test case for it. Clearly this is only valid in the context of the famous Dijkstra dictum: a test can show the presence of errors but can not guarantee their absence.

In general terms the tool takes as input a RSL file or files (this last is considering RSL modularity) and outputs the corresponding PVS file or files.

The tool is built in the general RSL type checker, which is a tool done and maintained at UNU/IIST [3] (and then follow the link 'tools').

The tool was constructed using the Gentle Compiler Construction System [8]. It is composed of several modules with a little more than ten thousand lines of Gentle code.

The Tool takes as input an RSL Abstract Syntax Tree produced by the RSL Type Checker and produces as output one or more .pvs files with the translation of the corresponding RSL when that is possible or one or more error messages.

8.2 Structure of the Tool

The general structure of the tool can be seen in Figure 3.

pvs.g is the main module that calls all the others and has the 'entrance' to the tool.

pvs_aux.g has auxiliary functions used by the other modules.

pvs_ast.g has the definition of PVS Abstract Grammar Since there are names in PVS Concrete Grammar for most non-terminals we have used when necessary the appropriate non-terminals names of the RSL Concrete Grammar.

pvs_col_sort.g sorts the declarations to comply with PVS "define before use".

pvs_gen_ast.g generates the PVS Abstract Syntax Tree from an RSL Abstract Syntax Tree.

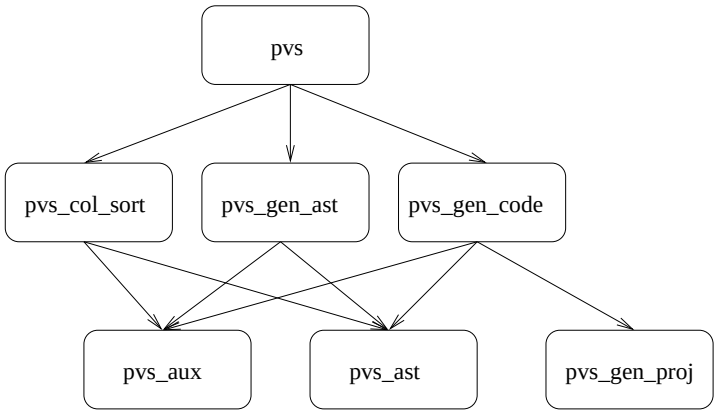

Figure 3: Tool Module Interdependency Graph

pvs_gen_code.g generates the PVS code (concrete grammar) from a PVS Abstract Syntax Tree.

pvs_gen_proj.g generates the PVS projections needed.

8.3 General Use

The Translator will produce .pvs files that can be submitted to the PVS tool for type checking and to the PVS Prover.

For **object**s or **scheme**s it will generate a .pvs file for each .rsl file with the name of the RSL file and extension pvs.

For RSL **theory** files it will generate one PVS `THEORY` for each class_scope_expr and group them in one .pvs file. It will do the same for **devt_relation**s making each RSL theory_expr a separate PVS `THEORY` and also group them in one .pvs file. The name of these files will be the same as the name used for the **theory** and the **devt_relation** respectively.

The names used in the RSL input specification for different entities (**type**s, **value**s, **axiom**s, etc.) are kept as much as possible

Some exceptions to the above rule are:

- names using Greek letters. Greek letters are translated as the string ≪`Rsl`≫ followed by the ASCII form of the Greek letter as shown in [6, Appendix C, page 384].
- names using primes. Primes are translated as the string ≪`rsL`≫.

- when there is an RSL binding or typing and it becomes necessary to create a PVS projection, then an special identifier ≪`proj_`≫ is generated and n is concatenated with it. Where n is an string representing a natural number also generated automatically as needed to be able to make the name unique.
- since in PVS all `AXIOM`s have to have an identifier this is the same as the optional RSL axiom_naming when is present. Otherwise an special identifier ≪`Ax_`≫ is generated and n is concatenated with it. Where n is an string representing a natural number also generated automatically, as needed, to be able to make the name unique. As the translator also generates axioms for some other purposes —implicit values, etc.— if you want to recognize your own RSL axioms then use the RSL optional-axiom_naming.

There are a number of RSL constructs that although they can be transformed into an equivalent PVS have not been implemented as yet in the tool:

- Patterns in **let** and **case** expressions are not fully implemented yet. There is a limited implementation for them. More details in Section 6.4.25, page 68 and in Section 6.4.27, page 70.
- Union definitions are not implemented.
- Reconstructors in Variants are not implemented.
- Object Declarations are not implemented, only Applicative Objects as modules are implemented.

PVS requires "define before use", so the .pvs file might look different from your RSL file if this rule has not been adhered to in RSL. It is just a cosmetic problem. If you want your PVS translated specification to have the same declaration order than the original RSL then keep this rule in mind.

The translator also alters the order of recursive function definitions: it puts them all at the end of the file. Again if you want your PVS translated specification to have the same declaration order than the original RSL then keep this rule in mind.

9 Concluding Remarks

Summary In this report we have explained the problems facing a transformation from one formal language into another. We have used RSL and PVS for our transformation.

We have explained the problems and what are some of the more important formal requierements that the transformation should have.

In adopting a practical approach for our translation and have gone systematically over every RSL language construct showing whether the construct can be transformed into a corresponding PVS and then providing a mapping into the semantic equivalent PVS construct or whether the construct can not be transformed and then we have said so and why.

We have also constructed a tool that implements the translation of nearly all the constructs that have been shown previously in the report that they can be translated.

The tool will not only provide a prover for RSL specifications but also it has helped to shape the whole transformation since it has forced at every step of its construction the need to check that what it was considered a secure transformation from RSL into PVS culd really be implemented and it really delivered —at least for a limited set of examples and case studies— the intended correct semantics. It has been a test bench for the whole transformation and we think it will continue to be so in the future.

Related Work There are in the literature references to similar kind of transformations or translations.

[9] verification of VDM and refinement with PVS similar to [10]

Agerholm in [10] [@@ and again Agerholm, Bicarregui and Maharajand in [9]] provide some manual heuristics —in the context of a "shallow embedding"— for a transformation of VDM into PVS. Only some very limited aspects of VDM are considered and a few heuristics are recomended. There are no claims to have covered the whole VDM language.

Buth in [11] [and also in citeButh98:tools] shows PAMELA using PVS for proofs. PAMELA is a system that "was originally developed for proving partial correctness of VDM specifications", but to find a solution to some drawbacks, PVS was chosen "to substitute a full proof system for the original simple proof component". This is a more automated approach than the one taken by Agerholm and one that creates an interface between PAMELA and PVS. This interface is built on a set of Tcl/Tk applications. The approach is also one of "shallow embedding". There are no claims to show a systematic treatment of how every construct of PAMELA maps into a corresponding semantically equivalent construct in PVS.

[12] ZF in HOL and Isabelle

[13] Code generation for State based formalisms

[14] method for formal method integration

[15] Isabelle for VDM-SL

[16] Proof Support for RAISE

[17] integration PVS and VDM

[18] comparison of PVS and Isabelle/HOL

[19] Z and HOL

Future Work One obvious future work that should be considered is completing the implementations in the tool of those RSL constructs that we have shown how they can be transformed into an equivalent PVS construct. Actually we can say that this is work in progress.

Since the complete $\mathcal{AG}_{\mathcal{RSL}}$ —and not a reduced one fitted for this transformation— was used, this leave open the door to further extend the transformation to more RSL constructs that are not covered now, prossibly concurrent RSL.

It will also be interesting to generate Confidence Conditions automatically and output them as a warning in the translator tool. An improvement over this will be to discharge the Confidence Conditions automatically.

Part III

Appendices

Summary. In this Part there are a few appendices with some additional useful information.

In Appendix A, page 102 there are cross reference tables relating the major RSL constructs as they appear in the RAISE Language Book [6] with their treatment in the report.

In Appendix B, page 106 there are tables that relate the operators of the two languages and their precedence and associativity.

A Index of RSL Constructs

This is an Index of the RSL Constructs taken from [6] and where in the Report they are treated.

Part I RSL Tutorial

Name (Chapter.Section)	Section in Report	Page in Report
Basic Concepts (3)		
Modules (3.2)	4	27
Type Declarations (3.3)	5.3	32
Value Declarations (3.4)	5.4	35
Axiom Declarations (3.5)	5.6	45
Module Extension (3.6)	4	27
Combining Value and Axiom Declarations (3.7)	5.4	35
Comments in Specifications (3.8)	7.8	91
Built-in Types (4)	3.1	13
Booleans (4.1)	3.1.1	13
If Expressions (4.1.1)	6.4.26	69
Prefix Connectives (4.1.2)	7.3.2	89
Infix Connectives (4.1.3)	7.3.1	88
Quantifiers (4.1.4)	6.4.11	61
Axiom Quantification (4.1.5)	Not used anymore in RSL	
Int (4.2)	3.1.1	14
Nat (4.3)	3.1.1	14
Real (4.4)	3.1.1	14
Characters (4.5)	3.1.1	15
Texts (4.6)	3.1.1	15
The Unit Value (4.7)	3.1.1	15
Products (5)		
Product Type Expressions (5.1)	6.3.3	47
Product Value Expressions (5.2)	6.4.5	53
Bindings and Typings (6)	7.9	91
Functions (7)	5.5.1, 5.5.2, 5.5.3, 5.5.4	37, 38, 40, 44
Predicative (7.10)	5.5.3	42
Algebraic (7.12)	5.5.3	42
Sets (8)	6.3.4	48
Lists (9)	6.3.5	49
Maps (10)	6.3.6	49
SubTypes (11)	6.3.7	50
Variant Definitions (12)	3.3	16
Case Expressions (13)	6.4.27	70
Let Expressions (14)	6.4.25	68
Union (15)	5.3.4	34
Under Specification and Non-determinism (16)	2	5

Part I RSL Tutorial

Name (Chapter.Section)	Section in Report	Page in Report
Overloading and User-defined Operators (17)	7.5	89
Variables and Sequencing (18)	5.7	45
Expressions Revisited (19)	6	46
Repetitive Expressions (20)	6.4.28, 6.4.29, 6.4.30	72, 72, 72
Local Expressions (21)	6.4.24	68
Algebraic Definition of Operations (22)	2.5	10
Post-expressions (23)	6.4.13	63
Channels and Communication (24)	5.7	45
Expressions Revisited (25)	6	46
Comprehended Expressions (26)	6	46
Algebraic Definition of Processes (27)	2.5	10
Modules (28)	4	27
Renaming and Hiding (29)	4	27
Parameterized Schemes (30)	4	27
Module Nesting (31)	4	27
Object Arrays (32)	4	27
The Name Space (33)	7.6	90

Part II RSL Reference Description

Name (Chapter.Section)	Section in Report	Page in Report
Declarative Constructs, Scope Rules and Visibility Rules (35)		
Declarative Constructs (35.1)	5	32
Scope Rules (35.2)	4	27
Visibility Rules (35.3)	4	27
Overloading (36)	7.5	89
Overload Resolution (36.2)	7.5	89
Specifications (37)	II	25
Declarations (38)	5	32
General (38.1)	4	27
Scheme Declarations (38.2)	4	27
Object Declarations (38.3)	4	27
Type Declarations (38.4)	5	32
Sort Definitions (38.4.1)	5	32
Variant Definitions (38.4.2)	3.3	16
Union Definitions (38.4.3)	5.3.4	34
Short Record Definitions (38.4.4)	3.3	16
Abbreviation Definitions (38.4.1)	5	32
Value Declarations (38.5)	5	32
Commented Typings (38.5.1)	7	73

Part II RSL Reference Description

Name (Chapter.Section)	Section in Report	Page in Report
Explicit Value Definitions (38.5.2)	5.4.2	36
Implicit Value Definitions (38.5.3)	5.4.3	36
Explicit Function Definitions (38.5.4)	5.5.1	37
Implicit Function Definitions (38.5.5)	5.5.2	38
Variable Declarations (38.6)	5.7	45
Channel Declarations (38.7)	5.7	45
Axiom Declarations (38.6)	5.6	45
Class Expressions (39)	4	27
Object Expressions (40)	4	27
Type Expressions (41)	6.3	46
General (41.1)	6.3	46
Type Literals (41.2)	3.1.1	13
Names (41.3)	6.3.1	46
Product Type Expressions (41.4)	6.3.3	47
Set Type Expressions (41.5)	6.3.4	48
List Type Expressions (41.6)	6.3.5	49
Map Type Expressions (41.7)	6.3.6	49
Function Type Expressions (41.8)	6.3.2	47
Subtype Expressions (41.9)	6.3.7	50
Bracketed Type Expressions (41.10)	6.3.8	50
Access Descriptions (41.11)	6.3.9	51
Value Expressions (42)	6.4	51
Value Literals (42.2)	6.4.1	51
Names (42.3)	6.4.2	53
Prenames (42.4)	6.4.3	53
Basic Expressions (42.5)	6.4.4	53
Product Expressions (42.6)	6.4.5	53
Set Expressions (42.7)	6.4.6	54
Ranged Set Expressions (42.7.1)	6.4.6	54
Enumerated Set Expressions (42.7.2)	6.4.6	54
Comprehended Set Expressions (42.7.3)	6.4.6	55
List Expressions (42.8)	6.4.7	55
Ranged List Expressions (42.8.1)	6.4.7	56
Enumerated List Expressions (42.8.2)	6.4.7	56
Comprehended List Expressions (42.8.3)	6.4.7	57
Map Expressions (42.9)	6.4.8	58
Enumerated Map Expressions (42.9.1)	6.4.8	58
Comprehended Map Expressions (42.9.2)	6.4.8	59
Function Expressions (42.10)	6.4.9	59
Application Expressions (42.11)	6.4.10	60
Quantified Expressions (42.12)	6.4.11	61
Equivalence Expressions (42.13)	6.4.12	63

Part II RSL Reference Description

Name (Chapter.Section)	Section in Report	Page in Report
Post Expressions (42.14)	6.4.13	63
Disambiguation Expressions (42.15)	6.4.14	64
Bracketed Expressions (42.16)	6.4.15	65
Infix Expressions (42.17)	6.4.16	65
Statement Infix Expressions (42.17.1)	6.4.16	65
Axiom Infix Expressions (42.17.2)	6.4.16	65
Value Infix Expressions (42.17.3)	6.4.16	65
Prefix Expressions (42.18)	6.4.17	66
Axiom Prefix Expressions (42.18.1)	6.4.17	66
Universal Prefix Expressions (42.18.2)	6.4.17	66
Value Prefix Expressions (42.18.3)	6.4.17	67
Comprehended Expressions (42.19)	6.4.18	67
Initialize Expressions (42.20)	6.4.19	67
Assignment Expressions (42.21)	6.4.20	67
Input Expressions (42.22)	6.4.21	67
Output Expressions (42.23)	6.4.22	68
Structured Expressions (42.24)	6.4.23	68
Local Expressions (42.24.1)	6.4.24	68
Let Expressions (42.24.2)	6.4.25	68
If Expressions (42.24.3)	6.4.26	69
Case Expressions (42.24.4)	6.4.27	70
While Expressions (42.24.5)	6.4.28	72
Until Expressions (42.24.6)	6.4.29	72
For Expressions (42.24.6)	6.4.30	72
Bindings (43) and Typings(44)	7.9	91
Patterns (45)	6.4.25	68
	6.4.27	70
Names (46)	7.6	90
Qualified Identifiers (46.2)	7.6.1	90
Qualified Operators (46.3)	7.6.1	90
Identifiers and Operators (47)	7.1, 7.2	73, 74
Infix Operators (47.2)	7.2.1	74
Prefix Operators (47.3)	7.2.2	84
Connectives (48)	7.3	88
Infix Connectives (48.1)	7.3.1	88
Prefix Connectives (48.2)	7.3.2	89
Infix Combinators (49)	7.4	89

B Operators. Precedence and Associativity Tables

Table 8: Precedence and Associativity (RSL and PVS)

	RSL		PVS	
Prec	Operators	Assoc	Operators	Assoc
21			FORALL, EXISTS, LAMBDA, IN	None
20			|	Left
19			|- , |=	Right
18			IFF, <=>	Right
17			IMPLIES, =>, WHEN,	Right
16			OR, \/, XOR, ORELSE	Right
15			AND, &, &&, /\, ANDTHEN	Right
14	□ λ ∀ ∃ ∃!	Right	NOT, ∼	None
13	≡ **post**		=, /=, ==, <, <=, >, >=, <<, >>, <<=, >>=, <|, |>	Left
12	[] ⫛ ∥ ⫴	Right	WITH	Left
11	;	Right	WHERE	Left
10	:=		@, #	Left
9	⇒	Right	@@, ##, ||	Left
8	∨	Right	+, -, ++	Left
7	∧	Right	, /, **, //	Left
6	= ≠ > < ≥ ≤ ⊂ ⊆ ⊃ ⊇ ∈ ∉		-	None
5	+ − \ ˆ ∪ †	Left	o	Left
4	* / ° ∩	Left	:, ::, HAS_TYPE	Left
3	↑		[], <>	Left
2	:		ˆ, ˆˆ	Left
1	∼ prefix_op		‘	Left

Table 9: RSL and corresponding PVS

Precedence for Individual Operators in RSL and corresponding PVS					
RSL			PVS		
Prec	Operators	Assoc	Prec	Operators	Assoc
14	$\Box$	Right			
14	λ	Right	21	`LAMBDA`	None
14	$\forall$	Right	21	`FORALL`	None
14	$\exists$	Right	21	`EXISTS`	None
14	$\exists!$	Right	21	no	
13	$\equiv$		18	`<=>`	Right
13	**post**				
12	$[]$	Right			
12	$\prod$	Right			
12	$\parallel$	Right			
12	$\#$	Right			
11	;	Right			
10	$:=$				
9	$\Rightarrow$	Right	17	`IMPLIES, =>`	Right
8	$\vee$	Right	16	`OR, \/, XOR, ORELSE`	Right
7	$\wedge$	Right	15	`AND, &, &&, /\`	Right
6	$= \neq > < \geq \leq$		13	`=, /=, <, <=, >, >=`	Left
6	$\subset \subseteq \supset \supseteq$			`subset()`, etc.	
6	$\in \notin$			`member()`, etc.	
5	$+ \quad -$	Left	8	`+, -`	Left
5	$\setminus$	Left		`difference()`	
5	$\hat{}$	Left		`append()`	
5	$\cup$	Left		`union()`	
5	$\dagger$	Left		`override()`[a]	
4	$* \quad /$	Left	7	`*, /`	Left
4	$\circ$	Left	5	`o`	Left
4	$\cap$	Left		`intersection()`	
3	$\uparrow$		2	`expt(), ^`	
2	:		4	:	Left
1	$\sim$		14	`NOT`, $\sim$	None
	prefix_op			functions	

All PVS functions in Prelude unless indicated otherwise

[a] in Map Theory

References

[1] S. Owre, N. Shankar, J. M. Rushby, and D. W. Stringer-Calvert. PVS Language Reference. Technical report, SRI International, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo PArk CA 94025, USA, September 1999. Version 2.3.

[2] The RAISE Method Group. *The RAISE Development Method*. Prentice Hall International (UK), 1995.

[3] International Institute for Software Technology. United Nations University. http://www.iist.unu.edu.

[4] Martin D. Fraser, Kuldeep Kumax, and Vijay K. Vaishnavi. Strategies for Incorporating Formal Specifications. *Communications of the ACM*, 37(10):74–86, October 1994.

[5] Chris George. RAISE Tools User Guide. Technical Report 227, UNU/IIST, P.O. Box 3058, Macau, February 2001.

[6] The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall International (UK), 1992.

[7] Natarajan Shankar and Sam Owre. Principles and pragmatics of subtyping in PVS. In Didier Bert, Christine Choppy, and Peter Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT '99*, volume 1827 of *Lecture Notes in Computer Science*, pages 37–52, Toulouse, France, September 1999. Springer-Verlag.

[8] *Gentle Compiler Construction System*. http://www.first.gmd.de/gentle/.

[9] Sten Agerholm, Juan Bicarregui, and Savi Maharaj. On the verification of VDM specification and refinement with PVS. In Juan Bicarregui, editor, *Proof in VDM: Case Studies*, FACIT (Formal Approaches to Computing and Information Technology), chapter 6, pages 157–190. Springer-Verlag, London, UK, 1997.

[10] Sten Agerholm. Translating specifications in VDM-SL to PVS. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, volume 1125 of *Lecture Notes in Computer Science*, pages 1–16, Turku, Finland, August 1996. Springer-Verlag.

[11] Bettina Buth. PAMELA + PVS. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 560–562, Sydney, Australia, December 1997. Springer-Verlag.

[12] Sten Agerholm and Mike Gordon. Experiments with ZF Set Theory in HOL and Isabelle. Technical report, BRICS, Department of Computer Science, University of Aarhus, 1995. http://www.brics.aau.dk/BRICS/.

[13] Michael Whalen. High Integrity Code Generation for State-Based Formalism. Technical report, Department of Computer Science and Engineering, University of Minnesota, 2000. Draft of paper submitted to ICSE 2000.

[14] Richard Paige. A Meta-Method for Formal Method Integration. Technical report, Department of Computer Science and Engineering, University of Toronto, 1998?

[15] Sten Agerholm and Jacob Frost. An Isabelle-based Theorem Prover for VDM-SL. In *Proceedings of TPHOLs97*. Springer, August 1997.

[16] Morten Lindegaard. Proof Support for RAISE, August 2001. Informatics and Mathematical Modelling, Technical University of Denmark.

[17] Georg Droschl. On the Integration of Formal Methods: Events and Scenarios in PVS and VDM. Technical report, Austrian Research Centre Seibersdorf and Technical University of Graz, March 1999.

[18] David Griffioen and Marieke Huisman. A comparison of PVS and Isabelle/HOL. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*, pages 123–142, Canberra, Australia, September 1998. Springer-Verlag.

[19] Jonathan Bowen and Mike Gordon. Z and HOL. Technical report, Oxford University and University of Cambridge, ?