SRI International

CSL Technical Report SRI-CSL-01-01 • April 2001

Theory Interpretations in PVS

Sam Owre and N. Shankar

Funded by NASA Langley Research Center contract numbers NAS1-20334 and NAS1-0079 and DARPA/AFRL contract number F33615-00-C-3043.

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

Abstract

We describe a mechanism for theory interpretations in PVS. The mechanization makes it possible to show that one collection of theories is correctly interpreted by another collection of theories under a user-specified interpretation for the uninterpreted types and constants. A theory instance is generated and imported, while the axiom instances are generated as proof obligations to ensure that the interpretation is valid. Interpretations can be used to show that an implementation is a correct refinement of a specification, that an axiomatically defined specification is consistent, or that a axiomatically defined specification captures its intended models.

In addition, the theory parameter mechanism has been extended with a notion of *theory as parameter* so that a theory instance can be given as an actual parameter to an imported theory. Theory interpretations can thus be used to refine an abstract specification or to demonstrate the consistency of an axiomatic theory. In this report we describe the mechanism in detail. This extension is a part of PVS version 3.0, which will be publicly released in mid-2001.

Contents

Chapter 1

Introduction

Theory interpretations have a long history in first-order logic [Sho67, End72, Mon76]. They are used to show that the language of a given source theory S can be interpreted within a target theory T such that the corresponding interpretation of axioms of S become theorems of T. This demonstrates the consistency of S relative to T, and also the decidability of S modulo the decidability of T. Theories and theory interpretations have also become important in higher-order logic and type theory with languages such as EHDM [EHD93], IMPS [Far92], HOL [Win92], Maude [CDE+99], Extended ML [ST97], and SPECWARE [SJ95]. In these languages, theories are used as structuring mechanisms for large specifications so that abstract theories can be refined into more concrete ones through interpretation. In this report, we describe a theory interpretation mechanism for the PVS specification language.

Specification languages and programming languages usually have some mechanism for packaging groups of definitions into modules. Lisp and Ada have *packages*. Standard ML has a module system consisting of signatures, structures corresponding to a signature, and functors that map between structures. Packages can be made generic by allowing certain declarations to serve as parameters that can be instantiated when the package is imported. Ada has *generic* packages that allow parameters. SML *functors* can be used to construct parametric modules. C++ allows *templates*.

In specification languages, a *theory* groups together related declarations of constants, types, axioms, definitions, and theorems. One way of demonstrating the consistency of such a theory is by providing an interpretation for the uninterpreted types and constants under which the axioms are valid. The definitions and theorems corresponding to a valid interpretation can then be taken as valid without further proof as long as they have been verified in the source theory. The technique of interpreting one axiomatic theory in another has many uses. It can be used to demonstrate the consistency or decidability of the former theory with respect to the latter theory. It can also be used to refine an abstract theory down to an executable implementation.

Interpretations are also useful in showing that the axioms capture the intended models. For example, a clock synchronization algorithm was developed in EHDM and was later shown to be consistent using the mappings, but it turned out that in one place $<$ was used instead of $\leq$, and because of this a set of perfectly synchronized clocks was actually disallowed by the model. Using interpretations in this way is similar to testing in allowing for the exploration of the space of models for the theory.

Parametric theories in PVS share some of the features of theory interpretations. Such theories can be defined with formal parameters ranging over types and individuals, for example,[1]

```
group[G: TYPE, + : [G, G -> G], 0: G, -: [G -> G]]: THEORY
  BEGIN
    ⋮
  END group
```

An instance of the theory `group` can be imported by supplying actual parameters, the type `int` of integers, integer addition `+`, zero `0`, and integer negation `-`, corresponding to the formal parameters, as in `group[int, +, 0, -]`. A theory can include assumptions about the parameters that have to be discharged when the actual parameters are supplied. For example, the group axioms can be given as assumptions in the `group` theory above. However, there are some crucial differences between parametric theories and theory interpretations. In particular, if axioms are always specified as assumptions, then the theory can be imported only by discharging these assumptions. It is necessary to have separate mechanisms for importing a theory with the axioms, and for interpreting a theory by supplying a valid interpretation, that is, one that satisfies its axioms.

The PVS theory interpretation mechanism is quite similar to that for theory parameterization. The axiomatic specification of groups could alternately be given in a theory

```
group: THEORY
 BEGIN
  G: TYPE+
  +: [G, G -> G]
  0: G
  -: [G -> G]
   ⋮
 END group
```

The group axioms are declared in the body of the theory. Such a theory can be interpreted by writing `group{{G := int, + := +, 0 := 0, - := -}}`. Here the left-hand sides refer to the uninterpreted types and constants of theory `group`, and the right-hand sides are the interpretations. This notation resembles that of theory parameterization

[1]This exploits a new feature of PVS version 3.0, in which numbers may be overloaded as names.

and is used in contexts where a theory is imported. The corresponding instances of the group axioms are generated as proof obligations at the point where the theory is imported. The result is a theory that consists of the corresponding mapping of the remaining declarations in the theory `group`. This allows the theory `group` to be used in other theories, such as rings and fields, and also allows the theory `group` to be suitably instantiated by group structures.

Theory interpretations largely subsume parametric theories in the sense that the theory parameters and the corresponding assumings can instead be presented as uninterpreted types and constants and axioms so that the actual parameters are given by means of an interpretation. However, a parametric theory with both assumings and axioms involving the parameters is not equivalent to any interpreted theory, as the parameters may be instantiated without the need to prove the axioms. It is also useful to have parametric theories as a convenient way of grouping together all the parameters that must be provided whenever the theory is used. For example, typical theory parameters such as the size of an array, or the element type of an aggregate datatype such as an array, list, or tree, are required as inputs whenever the corresponding theories are used. While this kind of parameterization can be captured by theory interpretations, it would not capture the intent that these parameters are *required* inputs wherever the theory is used. Furthermore, when an operation from a parametric theory is used, PVS attempts to figure out the actual parameters based on the context of its use. It can do this because the formal parameters are precisely delimited. The corresponding inference is harder for theory interpretations since there might be many possible interpretations that are compatible with the context of the operations use.

In addition to the uninterpreted types and constants in a source theory S, the PVS theory interpretation mechanism can also be used to interpret any theories that are imported into S by means of the `THEORY` declaration. The interpretation of a theory declaration for S' imported within S must itself be a theory interpretation of S'. Two distinct importations of a theory S' within S can be given distinct interpretations. A typical situation is when two theories R_1 and R_2 both import a theory S as S_1 and S_2, respectively. A theory T importing both R_1 and R_2 might wish to identify S_1 and S_2 since, otherwise, these would be regarded as distinct within T. This can be done by importing an instance S' of S into T and importing R_1 with S_1 interpreted by S' and R_2 with S_2 interpreted as S'. With theory interpretations, we have also extended parametric theories in PVS to take theories as parameters. For example, we might have a theory `group_homomorphism` of group homomorphisms that takes two groups `G1` and `G2` as parameters as in the declaration

```
group_homomorphism[G1, G2: THEORY group]: THEORY ...
```

The actual parameters for these theory formals must be interpretations `G1'` and `G2'` of the theory `group`.

Another typical requirement in a theory interpretation mechanism is the ability to map a source type to some quotient with respect to an equivalence relation over a target type. For example, rational numbers can be interpreted by means of a pair of integers corresponding to the numerator and denominator, but the same rational number can have multiple such representations. We show how it is possible to define quotient types in PVS and use these types to capture interpretations where the equality over a source type is mapped to an equivalence relation over a target type.

The implementation of theory interpretation in PVS is described in the following chapters. This report assumes the reader is already familiar with the PVS language; for details see the PVS Language Manual [OSRSC99]. Chapter 2 deals with mappings, explaining the basic concepts and introduces the grammar. Chapter 3 introduces theory declarations and theories as parameters which allow any valid interpretation of the formal parameter theory as an actual parameter. Chapter 4 describes a new command for viewing theory instances. Chapter 5 compares PVS interpretations with other systems, Chapter 6 describes future work, and we conclude with Chapter 7.

Chapter 2

Mappings

Theory interpretations in PVS provide mappings for uninterpreted types and constants of the *source* theory into the current (*interpreting*) theory. Applying a mapping to a source theory yields an *interpretation* (or *target*) theory. A mapping is specified by means of the *mapping* construct, which associates uninterpreted entities of the source theory with expressions of the target theory. The mapping construct is an extension to the PVS notion of "name". The changes to the existing grammar are given in Figure 2.1.

The mapping construct defines the basic translation, but to be a theory interpretation the mapping must be consistent: if type `T` is mapped to the type expression *E*, then a constant `t` of type `T` must be mapped to an expression *e* of type *E*. In addition, all axioms and theorems of the source theory must be shown to hold in the target theory under the mapping. Since the theorems are provable from the axioms, it is enough to show that the translation of the axioms hold. Axioms whose translations do not involve any uninterpreted types or constants of the source theory are converted to proof obligations. Otherwise they remain axioms.

Theory interpretation may be viewed as an extension of theory parameterization. Given a theory named `T`, the instance $\mathtt{T[a_1,...,a_n]\{\{c_1:= e_1,...,c_m:= e_m\}\}}$ is the same as the original theory, with the *actuals* $\mathtt{a}_i$ substituted for the corresponding formal

TheoryName	::=	[*Id* `@`] *Id* [*Actuals*] [*Mappings*]
Name	::=	[*Id* `@`] *IdOp* [*Actuals*] [*Mappings*] [`.` *IdOp*]
Mappings	::=	`{{` *Mapping*++'`,`' `}}`
Mapping	::=	*MappingLhs MappingRhs*
MappingLhs	::=	*IdOp Bindings*∗ [`:` { `TYPE` | `THEORY` | *TypeExpr* }]
MappingRhs	::=	`:=` { *Expr* | *TypeExpr* }

Figure 2.1: Grammar for Names with Mappings

parameters, and e_i substituted for c_i, which must be an uninterpreted type or constant declaration. Declarations that appear in the target of a substitution in the mapping are not visible in the importing theory. Some axioms are translated to proof obligations. The substituted forms of any remaining axioms, definitions, and lemmas are available for use, and are considered proved if they are proved in the uninterpreted theory.

The following simple example illustrates the basic concepts.

```
th1[T: TYPE, e: T]: THEORY
 BEGIN
  t: TYPE+
  c: t
  f: [t -> T]
  ax: AXIOM EXISTS (x, y: t): f(x) /= f(y)
  lem1: LEMMA EXISTS (x:T): x /= e
 END th1
```

```
th2: THEORY
 BEGIN
  IMPORTING th1[int, 0]
              {{ t := bool,
                 c := true,
                 f(x: bool) := IF x THEN 1 ELSE 0 ENDIF }}
  lem2: LEMMA EXISTS (x:int): x /= 0
 END th2
```

Here theory `th1` has both actual parameters and uninterpreted types and constants, as well as an axiom and a lemma. Theory `th2` imports `th1`, making the following substitutions:

```
T  ←  int
e  ←  0
t  ←  bool
c  ←  true
f  ←  LAMBDA (x: bool): IF x THEN 1 ELSE 0 ENDIF
```

Note that the mapping for `f` uses an abbreviated form of substitution. Typechecking this leads to the following proof obligation.

```
IMP_th1_TCC1: OBLIGATION
  EXISTS (x, y: bool):
    IF x THEN 1 ELSE 0 ENDIF /= IF y THEN 1 ELSE 0 ENDIF;
```

This is simply the interpretation of the `ax` axiom and is easily proved. The lemma `lem1` can be proved from the axiom, and may be used directly in proving `lem2` using the proof command `(LEMMA "lem1")`.

Note that once the TCC has been proved, we know that `th1` is consistent. If we had left out the mapping for `f`, then the TCC would not be generated, and the translation of theory `th1` would still contain an axiom and not necessarily be consistent.

One advantage to using mappings instead of parameters is that not all uninterpreted entities need be mapped, whereas for parameters either all or none must be given. For example, consider the following theory.

```
example1[T: TYPE, c: T]: THEORY
 BEGIN
  f(x: T): int = IF x = c THEN 0 ELSE 1 ENDIF
 END example1
```

It may be desirable to import this where `T` is always `real`, and `c` is left as a parameter, but there is currently no mechanism for this. One could envision partial importings such as `IMPORTING example1[real, _]`, but it is not clear that this is actually practical—in particular, the syntax for providing the missing parameters is not obvious. With mappings, on the other hand, we can define `example1` as follows.

```
example1: THEORY
 BEGIN
  T: TYPE
  c: T
  f(x: T): int = IF x = c THEN 0 ELSE 1 ENDIF
 END example1
```

Then we can refer to this theory from another theory as in the following.

```
example2: THEORY
 BEGIN
  th: THEORY = example1{{T := real}}
  frm: FORMULA f{{c := 3}} = f
 END example2
```

The `th` theory declaration just instantiates `T`, leaving `c` uninterpreted. The first reference to `f` maps `c` to `3`, whereas the second reference leaves it uninterpreted though it is still a `real`. Note that formula `frm` is unprovable, since the uninterpreted `c` from the second reference may or may not be equal to `3`.

As described in the introduction, an important aspect of mappings is the support for quotient types. In EHDM this was done by interpreting equality, but in PVS we instead define a theory of equivalence classes, and allow the user to map constants to equivalence classes under congruences. For example, the `stacks` datatype might be implemented using an array as follows.

```
stack[t:TYPE]: DATATYPE
 BEGIN
  empty: empty?
  push(top:t, pop: stack): nonempty?
 END stack
```

```
cstack[t: TYPE+]: THEORY
 BEGIN
  cstack: TYPE = [# size: nat, elems: [nat -> t] #]
  cempty?(s: cstack): bool = (s`size = 0)
  cempty: (cempty?) =
    (# size := 0,
       elems := LAMBDA (n: nat): epsilon(LAMBDA (x:t): true) #)
  cnonempty?(s: cstack): bool = (s`size /= 0)
  ctop(s: (cnonempty?)): t = s`elems(s`size - 1)
  cpop(s: (cnonempty?)): cstack = s WITH [`size := s`size - 1]
  cpush(x: t)(s: cstack): (cnonempty?) =
    (# size := s`size + 1,
       elems := s`elems WITH [(s`size) := x] #)
  ce(s1, s2: cstack): bool =
     s1`size = s2`size AND
     FORALL (n: below(s1`size)): s1`elems(n) = s2`elems(n)
  IMPORTING equivalence_class[cstack, ce], lifteq, lifteqs
  ...
```

The `equivalence_class` theory defines the quotient type of `cstack` with respect to the equivalence relation `ce`. It is defined as follows.

```
equivalence_class[T:TYPE, ==: (equivalence?[T])] : THEORY
 BEGIN
  x, y: VAR T

  equiv_class(x): setof[T] = {y | x == y}
  E: TYPE = {A: setof[T] | EXISTS x: A = equiv_class(x)}
  rep(A: E): (A) = epsilon(A)
  CONVERSION equiv_class, rep

  equiv_class_covers: LEMMA FORALL x: EXISTS (A: E): member(x, A)
  equiv_class_separates: LEMMA
      NOT (x == y)
        IMPLIES disjoint?(equiv_class(x), equiv_class(y))
 END equivalence_class
```

Note that it introduces `equiv_class` and `rep` as conversions. The type of the == parameter ensure that only equivalence relations are used in generating equivalence classes. The type `E` is the type of equivalence classes.

The `lifteq` and `lifteqs` theories allow functions on concrete stacks to be lifted to functions on equivalence classes, so long as they are congruences, that is, they satisfy the `preserves` relation.

```
lifteq[D, R: TYPE, deq: (equivalence?[D])]: THEORY
 BEGIN
  IMPORTING equivalence_class
  lift(f:(preserves[D, R](deq, =[R])))(A:E[D,deq]) : R
     = f(rep(A))
  CONVERSION lift
 END lifteq

lifteqs[D, R: TYPE,
        deq: (equivalence?[D]), req: (equivalence?[R])]
   : THEORY
 BEGIN
  IMPORTING equivalence_class
  lift(f:(preserves[D, R](deq, req)))(A:E[D,deq]) : E[R,req]
     = equiv_class[R,req](f(rep(A)))
  CONVERSION lift
 END lifteqs
```

For `lifteqs`, `f` satisfies the `preserves` relation if the following holds

```
FORALL (x1, x2: D): deq(x1,x2) IMPLIES req(f(x1),f(x2))
```

The reader might notice that the `lifteq` theory is not really necessary, as `lifteq[D, R, deq]` is semantically equivalent to `lifteqs[D, R, deq, =[R]]`. However, in practice the `lift` conversion of `lifteqs` is not applied without explicitly importing the correct instances. In addition, terms such as `rep[int,=[int]](equiv_class[int,=[int]](13))` end up being constructed, and it takes some work to reduce this to `13`.

With these theories imported, we can finish the specification of `cstack` as follows.

```
  ...
  estack: TYPE = E
  IMPORTING stack[t]{{ stack := estack,
                       empty? := cempty?,
                       nonempty? := cnonempty?,
                       empty := cempty,
                       push(x: t, s: estack) := cpush(x)(s),
                       top := ctop,
                       pop := cpop }}
 END cstack
```

Here the source type `stack` is mapped to the equivalence class `E` defined by the concrete equality `ce`, by means of the `equiv_class` conversion. The constant `empty` is then mapped to its equivalence class. The mapping for `push` is more involved; `cpush` must first be lifted in order to apply it to the abstract stack `s`. This is applied automatically by the conversion mechanism of PVS. The application of `lift` generates the proof obligation that `cpush` preserves the equivalences, that is, it is a congruence. This mapping generates a large number of proof obligations, because the `stack` datatype generates a `stacks_adt` theory with a large number of axioms, for example, extensionality, well-foundedness, and induction.

The PVS interpretations mechanism is much simpler to implement than the one in EHDM—equality is not a special case, but simply an aspect of mapping a type to an equivalence class. The technique of mapping types to equivalence classes is quite useful, and captures the notion of behavioral equivalence outlined in [ST97]. In fact it is more general, in that it works for any equivalence relation, not just those based on observable sorts.

Chapter 3

Theory Declarations

With the mapping mechanism, it is easy to specify a general theory and have it stand for any number of instances. For example, groups, rings, and fields are all structures that can be given axiomatically in terms of uninterpreted types and constants. This works well when considering one such structure at a time, but it is difficult to specify theories that involve more than one structure, for example, group homomorphisms. Importing the original theory twice is the same as importing it once, and an attempted definition of a homomorphism would turn into an automorphism. In this case what is needed is a way to specify multiple different "copies" of the original theory. This is accomplished with *theory declarations*, which may appear in either the theory parameters or the body of a theory. A theory declaration in the formal parameters is referred to as a *theory as parameter*.[1] Theory declarations allow theories to be encapsulated, and instantiated copies of the implicitly imported theory are generated.

For example, an (additive) group is normally thought of as a 4-tuple consisting of a set G, a binary operator $+$, an identity element 0, and an inverse operator $-$ that satisfies the usual group axioms. Using theory interpretations, we simply define this as follows:

[1]The term *theory parameter* refers to a parameter of a theory, so we use the term *theory as parameter* instead.

```
group: THEORY
 BEGIN
  G: TYPE+
  +: [G, G -> G]
  0: G
  -: [G -> G]
  x, y, z: VAR G
  associative_ax: AXIOM FORALL x, y, z: x + (y + z) = (x + y) + z
  identity_ax: AXIOM FORALL x: x + 0 = x
  inverse_ax: AXIOM FORALL x: x + -x = 0 AND -x + x = 0
  idempotent_is_identity: LEMMA x + x = x => x = 0
 END group
```

As described in Chapter 2, we can use mappings to create specific instances of groups. For example,

```
group{{G := int, + := +, 0 := 0, - := -}}
```

is the additive group of integers, whereas

```
group{{G := nzreal, + := *, 0 := 1, - := LAMBDA (r:nzreal): 1/r}}
```

is the multiplicative group of nonzero reals.

This works nicely, until we try to define the notion of a group homomorphism. At this point we need two groups, both individually instantiable. We could simply duplicate the group specification, but this is obviously inelegant and error prone. Using theories as parameters, we may define group homomorphisms as follows.

TheoryFormalDecl	::=	*TheoryFormalType* $\mid$ *TheoryFormalConst* $\mid$ *TheoryDecl*
TheoryDecl	::=	*Id* `:` `THEORY` *TheoryDeclName*
TheoryDeclName	::=	[*Id* `@`] *Id* [*Actuals*] [*TheoryDeclMappings*]
TheoryDeclMappings	::=	`{{` *TheoryDeclMapping*++ '`,`' `}}`
TheoryDeclMapping	::=	*MappingLhs TheoryDeclMappingRhs*
TheoryDeclMappingRhs	::=	*MappingSubst* $\mid$ *MappingDef* $\mid$ *MappingRename*
MappingSubst	::=	`:=` {*Expr* $\mid$ *TypeExpr*}
MappingDef	::=	`=` {*Expr* $\mid$ *TypeExpr*}
MappingRename	::=	`::=` {*IdOp* $\mid$ *Number*}

Figure 3.1: Grammar for Theory Declarations

```
group_homomorphism[G1, G2: THEORY group]: THEORY
 BEGIN
  x, y: VAR G1.G
  f: VAR [G1.G -> G2.G]
  homomorphism?(f): bool = FORALL x, y: f(x + y) = f(x) + f(y)
  hom_exists: LEMMA EXISTS f: homomorphism?(f)
 END group_homomorphism
```

Here `G1` and `G2` are theories as parameters to a generic homomorphism theory that may be instantiated with two different groups. Hence we may import `group_homomorphism`, for example, as

```
IMPORTING group_homomorphism[{{G := int, + := +, 0 := 0, - := -}}
                             {{G := nzreal, + := *, 0 := 1,
                                 - := LAMBDA (x: nzreal): 1/x}}]
```

There is a subtlety here that needs emphasizing; `G1` and `G2` are two *distinct* versions of theory `group`. For example, consider the addition of the following lemma to `group_homomorphism`.

```
oops: LEMMA G1.0 = G2.0
```

If `G1` and `G2` are treated as the same `group` theory, this is a provable lemma. But then after the importing given above we would be able to show that `0 = 1`. Even worse, the two different instances of groups may not even be type compatible, so the `oops` lemma should not even typecheck.

We have solved this in PVS by making new theories `G1` and `G2` that are copies of the original `group` theory. Declarations within these copies are distinct from each other and from the original. Thus the `oops` lemma generates a type error, as `G1.G` and `G2.G` are incompatible types.

This introduces new possibilities. When creating copies of a theory the mappings are substituted and the original declarations disappear. However, it may be preferable to create definitions rather than substitutions. In addition, it is sometimes useful to simply rename the types or constants of a theory. For example, consider the following group instance

```
G1: THEORY = group{{G := int, + := +, 0 := 0, - := -}}
```

which generates the following theory.

```
G1: THEORY
 BEGIN
  x, y, z: VAR int
  idempotent_is_identity: LEMMA x + x = x => x = 0
 END G1
```

To create definitions, use = instead of `:=`, as in the following.

```
G2: THEORY = group{{G = int, + = +, 0 = 0, - = -}}
```

Now we get the following theory.

```
G2: THEORY
 BEGIN
  G: TYPE+ = int
  +: [G, G -> G] = +
  0: G = 0
  -: [G -> G] = -
  x, y, z: VAR G
  idempotent_is_identity: LEMMA x + x = x => x = 0
 END G2
```

Finally, to simply rename the uninterpreted types and constants, use `::=` as in the following.

```
G3: THEORY = group{{G ::= MG, + ::= *, 0 ::= 1, - ::= inv}}
```

The generated theory instance specifies multiplicative groups as follows.

```
G3: THEORY
 BEGIN
  MG: TYPE+
  *: [MG, MG -> MG]
  1: MG
  inv: [MG -> MG]
  x, y, z: VAR MG
  associative_ax: AXIOM FORALL x, y, z: x * (y * z) = (x * y) * z
  identity_ax: AXIOM FORALL x: x * 1 = x
  inverse_ax: AXIOM FORALL x: x * inv(x) = 1 AND inv(x) * x = 1
  idempotent_is_identity: LEMMA x * x = x => x = 1
 END G3
```

The right-hand side of a renaming mapping must be an identifier, operator, or number, and must not create ambiguities within the generated theory. Note that renamed declarations are still uninterpreted, and may themselves be given interpretations, as in

```
G3i: THEORY = G3{{MG := nzreal, * := *, 1 := 1,
                  inv := LAMBDA (r: nzreal): 1/r}}
```

Finally, we can mix the different forms of mapping, to give a partial mapping.

```
G4: THEORY = group{{G = nzreal, + := *, 0 ::= one}}
```

This generates the following theory instance.

```
G4: THEORY
 BEGIN
  G: TYPE+ = nzreal;
  one: nzreal;
  -: [nzreal -> nzreal]
  x, y, z: VAR nzreal
  identity_ax: AXIOM FORALL (x: nzreal): x * one = x
  inverse_ax: AXIOM FORALL (x: nzreal):
                      x * -x = one AND -x * x = one
  idempotent_is_identity: LEMMA x * x = x => x = one
 END G4
```

Note that `associative_ax` has disappeared—it has become a TCC of the importing theory—whereas the other axioms are not so transformed because they still reference uninterpreted types or constants.

With theories as parameters we have another situation in which mappings are more convenient than theory parameters. Many times the same set of parameters is passed through an entire theory hierarchy. If there are assumings, then these must be copied. For example, consider the following theory.

```
th[T: TYPE, a, b: T]: THEORY
 BEGIN
  ASSUMING
   A: ASSUMPTION a /= b
  ENDASSUMING
  ...
 END th
```

To import this theory, you simply provide a type and two different elements of that type. But suppose you wish to import this theory from a theory that has the same parameters. In this case the assumption must also be copied, as there is otherwise no way to prove the resulting obligation. This can (and frequently does) lead to a tower of theories, all with the same parameters and copies of the same assumptions, as well as proofs of the same obligations.

There are ways around this, of course. Most assumptions may be turned into type constraints, as in the following.

```
th[T: TYPE, a: T, b: {x: T | a /= x}]: THEORY
 ...
```

But this introduces an asymmetry in that `a` and `b` now belong to different types, and the type predicate still must be provided up the entire hierarchy.

Using a theory as a parameter, we may instead define `th` as follows.

```
th: THEORY
 BEGIN
  T: TYPE,
  a, b: T
  A: AXIOM a /= b
  ...
 END th
```

We then parameterize using this theory (which is implicitly imported):

```
th_1[t: THEORY th]: THEORY ...
```

We have encapsulated the uninterpreted types and constants into a theory, and this is now represented as a single parameter. Axiom `A` is visible within theory `th_1`, and no proof obligations are generated since no mapping was given for `th`. Now we can continue defining new theories as follows.

```
th_2[t: THEORY th]: THEORY IMPORTING th_1[t] ...
th_3[t: THEORY th]: THEORY IMPORTING th_2[t] ...
  ⋮
```

None of these generate proof obligations, as no mappings are provided.

We may now instantiate `th_n`, for example, with the following.

```
IMPORTING th_n[th{{T := int, a := 0, b := 1}}]
```

Now the substituted form of the axiom becomes a proof obligation which, when proved, provides evidence that the theory `th` is consistent.

With the introduction of theories as parameters, it is natural to allow theory declarations that may be mapped, in the same way that instances may be provided for theories as parameters. Thus the `group_homomorphism` may be rewritten as follows:

```
group_homomorphism: THEORY
 BEGIN
  G1, G2: THEORY group
  x, y: VAR G1.G
  f: VAR [G1.G -> G2.G]
  homomorphism?(f): bool = FORALL x, y: f(x + y) = f(x) + f(y)
  hom_exists: LEMMA EXISTS f: homomorphism?(f)
 END group_homomorphism
```

Again, the choice between using theories as parameters or theory declarations is really a question of taste, as they are largely interchangeable.

As with theories as parameters, copies must be made for `G1` and `G2`. Note that this means that there is a difference between theory abbreviations and theory declarations, as

Importing	::=	`IMPORTING` *ImportingItem*++','
ImportingItem	::=	*TheoryName* [`AS` *Id*]

Figure 3.2: Grammar for Importings

the former do not involve any copying. We decided to use the old form of theory abbreviation to define theory declarations, and to extend the `IMPORTING` expressions to allow abbreviations, as shown in Figure 3.2. Thus instead of

```
funset: THEORY = sets[[int -> int]]
```

which creates a copy of sets, use

```
IMPORTING sets[[int -> int]] AS funset
```

which imports `sets[[int -> int]]` and abbreviates it as `funset`.

Chapter 4

Prettyprinting Theory Instances

Mappings can get fairly complex, especially if actual parameters are involved, and it may be desirable to see the specified theory instance displayed with all the substitutions performed. To support this, we have provided a new PVS command: `prettyprint-theory-instance` (`M-x ppti`). This takes two arguments: a theory instance, which in general is a theory name with actual parameters and/or mappings, and a context theory, in which the theory instance may be typechecked. The simplest way to use this command is to put the cursor on the theory name as it appears in a theory as parameter, theory declaration, or importing—when the command is issued it then defaults to the theory instance under the cursor and the current theory is the default context theory. For example, putting the cursor on `group_homomorphism` in the following and typing `M-x ppti` followed by two carriage returns[1] generates a buffer named `group_homomorphism.ppi`. All instances of a given theory generate the same buffer name.

```
IMPORTING group_homomorphism[{{G := int, + := +, 0 := 0, - := -}}
                             {{G := nzreal, + := *, 0 := 1,
                               - := LAMBDA (x: nzreal): 1/x}}]
```

This buffer has the following contents.

[1]The first uses the theory name instance at the cursor, and the second uses the current theory as the context.

```
% Theory instance for
  % group_homomorphism[groups{{ G := int, + := +,
  %                             - := -, 0 := 0 }},
  %                    groups{{ G := nzreal, + := *,
  %                             - := (LAMBDA (x: nzreal): 1 / x),
  %                             0 := 1 }}]
group_homomorphism_instance: THEORY
 BEGIN

  IMPORTING groups{{ G := int, + := +, - := -, 0 := 0 }}

  IMPORTING groups{{ G := nzreal, + := *,
                   - := (LAMBDA (x: nzreal): 1 / x), 0 := 1 }}

  x, y: VAR int

  f: VAR [int -> nzreal]

  homomorphism?(f): bool =
    FORALL (x: int), (y: int): f(x + y) = f(x) * f(y)

  hom_exists: LEMMA EXISTS (f: [int -> nzreal]): homomorphism?(f)
 END group_homomorphism_instance
```

The group instances shown on pages 13–15 provide more examples of the output produced by `prettyprint-theory-instance`.

Chapter 5

Comparison with Other Systems

In this chapter we compare PVS theory interpretations to existing programming and specification mechanisms of other systems. The EHDM system [EHD90] has a notion of a mapping module that maps a source module to a target module. When a mapping module is typechecked, a new module is automatically created that represents the substitution of the interpretations for the body of the source theory. Equality is allowed to be mapped in EHDM, in which case it must be mapped to an equivalence relation. In PVS, mappings are provided as a syntactic component of names, and are essentially an extension of theory parameters. Equality is not treated specially, but is handled by mapping a given type to a quotient type.

IMPS [FGT90, Far94] also supports theory interpretations. It is similar to EHDM in that it has a special `def-translation` form that takes a source theory, target theory, sort association list, and constant association list, and generates a theory translation. Obligations may be generated that ensure that every axiom of the source theory is a theorem of the target theory. If these are proved the translation is treated as an interpretation. There is no mechanism for mapping equality. As with both PVS and EHDM, defined sorts and constants of the source theory are automatically translated. A more detailed comparison between IMPS and an earlier version of PVS appears in an unpublished report by Kammüller [Kam96].

In Maude [CDE$^+$99] and its precursor OBJ [GW88] it is possible to define `modules` that represent transition systems of a rewrite theory whose states are equivalence classes of ground terms and whose transitions are inference rules in *rewriting logic*. A given module may import another module, either `protecting` it, which means that the importing module adds no *junk* or *confusion*, or `including` it, which imposes no such restrictions. In addition to modules, Maude has *theories*, which are used to declare module interfaces. These may appear as module parameters, as in $M[X_1 :: T_1, \ldots, X_n :: T_n]$, where the X_i are *labels* and the T_i are names of theories. These theory parameters (source theories) may be instantiated by target theories or modules using *views*, which indicate how each sort, function, class, and message of the source theory is mapped to the target theory. However, Maude currently does not support the generation of proof obligations from source theory axioms, so views are simply theory translations, not interpretations.

The programming language Standard ML [MTH90] has a module system where modules are given by *structures* with a given *signature*, and parametric modules are *functors* mapping structures of a given signature to structures. The PVS mechanism of using theories as parameters resembles SML functors but for a specification language rather than a programming language. Sannella and Tarlecki [ST97] describe a version of the ML module system in which there are *specifications* containing *sorts*, *operations*, and *axioms*. For example, the signature of stacks is the following.

$$
\begin{array}{lll}
\mathit{STACK} = & \textbf{sorts} & \mathit{stack} \\
 & \textbf{opns} & \mathit{empty} : \mathit{stack} \\
 & & \mathit{push} : \mathtt{int} \times \mathit{stack} \to \mathit{stack} \\
 & & \mathit{pop} : \mathit{stack} \to \mathit{stack} \\
 & & \mathit{top} : \mathit{stack} \to \mathtt{int} \\
 & & \mathit{is\_empty} : \mathit{stack} \to \mathtt{bool} \\
 & \textbf{axioms} & \mathit{is\_empty}(\mathit{empty}) = \mathtt{true} \\
 & & \forall s : \mathit{stack}. \forall n : \mathtt{int}. \mathit{is\_empty}(\mathit{push}(n,s)) = \mathtt{false} \\
 & & \forall s : \mathit{stack}. \forall n : \mathtt{int}. \mathit{top}(\mathit{push}(n,s)) = n \\
 & & \forall s : \mathit{stack}. \forall n : \mathtt{int}. \mathit{pop}(\mathit{push}(n,s)) = s
\end{array}
$$

The following algebra is a *realization* of the above specification that corresponds to that of `cstack` on page 8.

```
structure S2 : STACK =
    struct
        type stack = (int -> int) * int
        val empty = ((fn k => 0), 0)
        fun push (n, (f, i))
              = ((fn k => if k = i then n else f k), i+1)
        fun pop (f, i) = if i = 0 then (f, 0) else (f, i-1)
        fun top (f, i) = if i = 0 then 0 else f(i-1)
        fun is_empty (f, i) = (i=0)
```

Note however, that the stacks *empty* and *pop*(*push*(6,*empty*)) are not equal. Thus they distinguish the *observable* sorts, in this case `int` and `bool`, which are the only data directly visible to the user. The above two terms are not *observable computations*, so it does not matter that they are different. In general, two different algebras are *behaviorally equivalent* if all observable computations yield the same results. Note that choosing observable values based on sorts is a bit coarse: for example, there may be two `int`-valued variables, one of which is observable and one that represents an internal pointer. Mapping to equivalence classes is more general, as it is easy to capture behavioral equivalence.

The induction theorem prover Nqthm [BM88, BGKM91] has a feature called `FUNCTIONALLY-INSTANTIATE` that can be used to derive an instance of a theorem by supplying an interpretation for some of the function symbols used in defining the theorem. The corresponding instances of any axioms concerning these function symbols must be discharged. Such axioms can be introduced as conservative extensions as definitions with the `DEFUN` declaration or through witnessed constraints using the `CONSTRAIN` declaration, or they can be introduced nonconservatively through an `ADD-AXIOM` declaration. While the functional instantiation mechanism is similar in flavor to PVS theory interpretations, the underlying logic of Nqthm is a fragment of first-order logic whose expressive power is more limited than the higher-order logic of PVS. In addition, Nqthm lacks types and structuring mechanisms such as parametric theories.

The SPECWARE language [SJ95] employs theory interpretations as a mechanism for the stepwise refinement of specifications into executable code. SPECWARE has constructs for composing specifications while identifying the common components, and for compositionally refining specifications so that the refinement of a specification can be composed from the refinement of its components. Unlike PVS, SPECWARE has the ability to incorporate multiple logics and translate specifications between these logics. A theory is an independent unit of specification in PVS and hence there is no support for composing theories from other theories. However, the operations in SPECWARE can largely be simulated by means of theories and theory interpretations in PVS.

In summary, theory interpretation has been a standard tool in specification languages since the early work on HDM [RLS79] and Clear [BG81]. PVS implements theory interpretations as a simple extension of the mechanism for importing parametric theories. PVS theory interpretations subsume the corresponding capabilities available in other specification frameworks.

Chapter 6

Future Work

A number of interesting extensions may be contemplated for the future.

Mapping of interpreted types and constants— There are two aspects: one is simply a convenience where, for example, we might have a tuple type declaration `T: TYPE = [T1, T2, T3]` and want to map it to `position: TYPE = [real, real, real]` by simply giving the map `{{T := position}}`.

The second aspect is where the mapping is between two different kinds, for example mapping a record type to a function type. This requires determining the corresponding components as well as making explicit the underlying axioms. For example, record types satisfy extensionality, and if they are mapped to a different type the implicit extensionality axiom must be translated to a proof obligation.

Rewriting with congruences— In theory substitution, if a type is mapped to a quotient type then equality over this type is mapped to equality over the quotient type. If T is an uninterpreted type, $\equiv$ an equivalence relation over T', and $T'/\equiv$ the quotient type, then `=[`T`]` is mapped to `=[`$T'/\equiv$`]`, which is equivalent to $\equiv$. An equational formula thus still has the form of a rewrite. However, to apply such a rewrite one generally needs to do some lifting. The following is a simple example.

```
th: THEORY
 BEGIN
  T: TYPE
  a, b: T
  f, g: [T -> T]
  ... Some axioms involving f, g, a, and b
  lem: LEMMA f(a) = g(b)
 END th
th2: THEORY
 BEGIN
  ==(x, y: int): bool = divides(3, x - y)
  IMPORTING th{{T := E(==),
                a := equiv_class(==)(2),
                b := equiv_class(==)(1),
                f := LAMBDA (x: E(==)): equiv_class(rep(x) - 1),
                g := LAMBDA (x: E(==)): equiv_class(rep(x) - 2)}}
  ...
 END th2
```

To rewrite with `lem`, `a` must first be lifted to its equivalence class, then the rewrite is applied and the result is then projected back using `rep`. To do this requires some modification to the rewriting mechanism of the prover.

Consistency Analysis— With a single independent theory such as groups, it is easy to generate a mapping in which all axioms become proof obligations, and see directly that the theory is consistent. On the other hand, if many theories are involved in which compositions of mappings are involved, this may become quite difficult. What is needed is a tool that analyzes a mapped theory to see if it is consistent, and reports on any remaining axioms and uninterpreted declarations. This is similar in spirit to proof chain analysis, but works at the theory level rather than for individual formulas.

Semantics of Mappings— The semantics of theory interpretations needs to be formalized and added to the PVS semantics report [OS97].

Chapter 7

Conclusion

Theory interpretations are used to embed an interpretation of an abstract theory in a more concrete one. In this way, they allow an abstract development to be reused at the more concrete level. Theory interpretations can be used to refine a specification down to code. Theory interpretations can also be used to demonstrate the consistency of an axiomatic theory relative to another theory.

Parametric theories in PVS provide some but not all of the functionality of theory interpretations. In particular, they do not allow an abstract theory to be imported with only a partial parameterization. Theory interpretations have been implemented in PVS version 3.0, which will be released in mid-2001. The current implementation allows the interpretation of uninterpreted types and constants in a theory, as well as theory declarations. PVS has also been extended so that a theory may appear as a formal parameter of another theory. This allows related sets of parameters to be packaged as a theory. Quotient types have been defined within PVS and used to admit interpretations of types where the equality on a source type is treated as an equivalence relation on a target type.

Theory interpretations have been implemented in PVS as an extension of the theory parameter mechanism. This way, theory interpretations are an extension of an already familiar concept in PVS and can be used in place of theory parameters where there is a need for greater flexibility in the instantiation. The proof obligations generated by theory interpretations are similar to those for parametric theories with assumptions.

A number of extensions related to theory interpretations remain to be implemented. First, we plan to extend theory interpretations to the case of interpreted types and constants. This poses some challenges since there are implicit operations and axioms associated with certain type constructors. Second, the rewriting mechanisms of the PVS prover need to be extended to rewrite relative to a congruence. This means that if we are only interested in $f(a)$ up to some equivalence that is preserved by f, then we could rewrite a up to equivalence rather than equality. Third, the PVS semantics have to be extended to incorporate theory interpretations. Finally, the PVS ground evaluator has to be extended to handle theory interpretations. Currently, the ground evaluator generates code corresponding to a parametric theory and this code is reused with the actual parameters used as arguments to the operations. Theory interpretations cannot be treated as arguments in this manner since there is no fixed set of parameters; parameters can vary according to the interpretation. Also, non-executable operations can become executable as a result of the interpretation.

In summary, we believe that theory interpretations are a significant extension to the PVS specification language. Our implementation of this in PVS3.0 is simple yet powerful. We expect theory interpretations to be a widely used feature of PVS.

Bibliography

[BG81] R. M. Burstall and J. A. Goguen. An informal introduction to specifications using Clear. In *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[BGKM91] Robert S. Boyer, David M. Goldschlag, Matt Kaufmann, and J S. Moore. Functional instantiation in first-order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theorem of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

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

[CDE+99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. Technical Report CDRL A005, Computer Science Laboratory, SRI International, March 1999.

[EHD90] Computer Science Laboratory, SRI International, Menlo Park, CA. EHDM *Specification and Verification System Version 5.0—Description of the* EHDM *Specification Language*, January 1990. See [EHD91] for the updates to Version 5.2.

[EHD91] Computer Science Laboratory, SRI International, Menlo Park, CA. EHDM *Specification and Verification System Version 5.X—Supplement to User's and Language Manuals*, August 1991. Current version number is 5.2.

[EHD93] Computer Science Laboratory, SRI International, Menlo Park, CA. *User Guide for the* EHDM *Specification Language and Verification System, Version 6.1*, February 1993. Three volumes.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York, NY, 1972.

[Far92] William M. Farmer. Theory interpretations in computerized mathematics (abstract). *Journal of Symbolic Logic*, 57(1):356, March 1992.

[Far94] W. M. Farmer. Theory interpretation in simple type theory. In J. Heering et al., editor, *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 96–123. Springer-Verlag, 1994.

[FGT90] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction (CADE)*, volume 449 of *Lecture Notes in Computer Science*, pages 653–654, Kaiserslautern, Germany, July 1990. Springer-Verlag.

[GW88] Joseph A. Goguen and Timothy Winkler. Introducing OBJ. Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1988.

[Kam96] F. Kammüller. Comparison of IMPS, PVS and Larch with respect to theory treatment and modularization. Technical report, Computer Laboratory, University of Cambridge, 1996. Unpublished Draft 1.0, available at `http://www.first.gmd.de/~florian/papers/report.ps.gz`.

[Mon76] J. Donald Monk. *Mathematical Logic*. Graduate Texts in Mathematics. Springer-Verlag, New York, NY, 1976.

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

[OS97] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[OSRSC99] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[RLS79] L. Robinson, K. N. Levitt, and B. A. Silverberg. *The HDM Handbook*. Computer Science Laboratory, SRI International, Menlo Park, CA, June 1979. Three Volumes.

[Sho67] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, MA, 1967.

[SJ95] Yellamraju V. Srinivas and Richard Jüllig. Specware: Formal support for composing software. In Bernhard Möller, editor, *Mathematics of Program Construction*, number 947 in Lecture Notes in Computer Science, pages 399–422. Springer-Verlag, 1995.

[ST97] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[Win92] Phillip J. Windley. Abstract theories in HOL. In Luc Claesen and Michael J. C. Gordon, editors, *Proceedings of the 1992 International Workshop on the HOL Theorem Prover and its Applications*, pages 197–210, Leuven, Belgium, September 1992. IFIP, North–Holland.