Principles and Pragmatics of Subtyping in PVS[⋆]

Invited paper at the 1999 Workshop on Abstract Datatypes

Natarajan Shankar and Sam Owre

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{shankar, owre}@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (650) 859-5272 Fax: +1 (650) 859-2844

Abstract. PVS (Prototype Verification System) is a mechanized framework for formal specification and interactive proof development. The PVS specification language is based on higher-order logic enriched with features such as predicate subtypes, dependent types, recursive datatypes, and parametric theories. Subtyping is a central concept in the PVS type system. PVS admits the definition of subtypes corresponding to nonzero integers, prime numbers, injective maps, order-preserving maps, and even empty subtypes. We examine the principles underlying the PVS subtype mechanism and its implementation and use.

The PVS specification language is primarily a medium for communicating formal mathematical descriptions. Formal PVS specifications are meant for both machine and human consumption. The specification language of PVS extends simply typed higher-order logic with features such as predicate subtypes, dependent types, recursive datatypes, and parametric theories. These features are critical for facile mathematical expression as well as symbolic manipulation. Though the language has been designed to be used in conjunction with a theorem prover, it has an existence independent of any specific theorem prover.

The core specification language of PVS is quite small yet poses a number of serious implementation challenges. We outline the difficulties in realizing these features in a usable implementation. Our observations might be useful to designers and implementors of other specification languages with similar features.

There is a long history of formal foundational languages for mathematics. Frege's *Begriffsschrift* [Fre67a] was presented as a system of axioms and rules for logical reasoning in the sciences. Frege's use of function variables was found to be inconsistent by Russell [Rus67,Fre67b]. Poincare attributed the problem to a *vicious circle*, or impredicativity, that allowed an entity to be defined by quantification over a domain that included the entity itself. There were two initial responses to this. Zermelo's solution was to craft an untyped set theory where comprehension was restricted to extracting subsets of existing sets. Russell and

[⋆] This work was funded by NSF Grants No. CCR-9712383 and CCR-9509931.

Whitehead's system of *Principia Mathematica* [WR27] consisted of a simple theory of types which stratified the universe into the type of individuals, collections of individuals, collections of collections of individuals, etc., and a ramified theory of types that stratified the elements within a type to rule out impredicative definitions.[1]

In computing, specification languages are meant to formalize *what* is being computed rather than *how* it is computed. There are many discernible divisions across these specification languages including

- Set theory (Z [Spi88], VDM [Jon90]) *versus* type theory (HOL [GM93], Nuprl [CAB$^+$86], Coq [DFH$^+$91], PVS [ORSvH95])
- Constructive (Coq, Nuprl) *versus* classical foundations (Z, HOL, PVS)
- First-order (OBJ [FGJM85b], Maude [CDE$^+$99], VDM, CASL [Mos98][2]) *versus* higher-order logic (HOL, Nuprl, Coq, PVS)
- Model-oriented (Z, VDM) *versus* property-oriented (OBJ, Maude, CASL)
- Total function (HOL, PVS) *versus* partial function (OBJ, Maude, VDM, CASL)

The PVS specification language is based on a strongly typed higher-order logic of total functions that builds on Church's simply typed higher-order logic [Chu40,And86]. Higher-order logic captures only a modest fragment of set theory, but it is one that is reasonably expressive and yet effectively mechanizable. Types impose a useful discipline within a specification language. They also lead to the early detection of a large class of syntactic and semantic errors. PVS admits only total functions but this is mitigated by the presence of subtypes since a partial function can be introduced as a total function when its domain of definition can be captured as a subtype. For example, the division operation can be introduced with the domain given as the subtype of numbers consisting of the nonzero numbers. If applied to a term not known to be nonzero, a proof obligation is generated. PVS is based on a classical foundation as opposed to a constructive one since constructive proofs impose a substantial cost in proof construction for a modest gain in the information that can be extracted from a successful proof.

We focus primarily on subtyping and the surrounding issues since this is one of the core features of PVS. We also compare PVS with other specification languages. The paper condenses material from reports: *The Formal Semantics of PVS* [OS97] (URL: www.csl.sri.com/reports/postscript/csl-97-2.ps.gz) and *Abstract Datatypes in PVS* [OS93] (URL: www.csl.sri.com/reports/ postscript/csl-93-9.ps.gz). These reports should be consulted for further details. Rushby, Owre, and Shankar [ROS98] motivate the need for PVS-style subtyping in specification languages.

Following the style of the formal semantics of PVS [OS97], we present an idealized core of the PVS language in small increments by starting from the

[1] Rushby [Rus93] has a lengthy discussion of foundational issues and their impact on specification language features.

[2] CASL also has a higher-order extension.

simple type system, and adding predicate subtypes, dependent types, typing judgements, and abstract datatypes.

1 The Simply Typed Fragment

The *base types* in PVS consist of the booleans `bool` and the real number type `real`.[3] From types T_1 and T_2, a *function type* is constructed as $[T_1 \rightarrow T_2]$ and a *product type* is constructed as $[T_1, T_2]$.

The *preterms* t consist of

- constants: c, f
- variables: x
- pairs: $\langle t_1, t_2 \rangle$
- projections: $\mathtt{p}_i\ t$
- abstractions: $\lambda(x : T)\ t$
- applications: ft

The typechecking of a preterm a is carried out with respect to a declaration context Γ by an operation $\tau(\Gamma)(a)$ that returns the canonical type. A context is a sequence of bindings of names to types, kinds, and definitions. For an identifier s, $\mathit{kind}(\Gamma(s))$ returns the kind `CONSTANT`, `VARIABLE`, or `TYPE`. Since the contexts and types also need to be typechecked, we have

- $\tau()(\Gamma) = \mathtt{CONTEXT}$ for well-formed context Γ.
- $\tau(\Gamma)(A) = \mathtt{TYPE}$ for well-formed type A.

The definition of $\tau(\Gamma)(a)$ is

$$\begin{aligned}
\tau(\Gamma)(s) &= \mathit{type}(\Gamma(s)),\\
&\quad \text{if } \mathit{kind}(\Gamma(s)) \in \{\mathtt{CONSTANT}, \mathtt{VARIABLE}\}\\
\tau(\Gamma)(f\ a) &= B,\ \text{if } \tau(\Gamma)(f) = [A \rightarrow B] \text{ and } \tau(\Gamma)(a) = A\\
\tau(\Gamma)(\lambda(x : T)\ a) &= [T \rightarrow \tau(\Gamma, x : \mathtt{VAR}\ T)(a)],\ \text{if } \Gamma(x) \text{ is undefined}\\
&\quad \text{and } \tau(\Gamma)(T) = \mathtt{TYPE}\\
\tau(\Gamma)(\langle a_1, a_2 \rangle) &= [\tau(\Gamma)(a_1), \tau(\Gamma)(a_2))]\\
\tau(\Gamma)(\mathtt{p}_i\ a) &= T_i,\ \text{where}\\
&\quad \tau(\Gamma)(a) = [T_1, T_2]
\end{aligned}$$

Type rules [Car97] are conventionally given as inference rules of the form

$$\frac{\Gamma \vdash f : [A \rightarrow B] \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$$

[3] The actual base number type is an unspecified supertype of the reals called `number` which is there to accommodate possible extensions of the reals such as the extended reals (with $+\infty$ and $-\infty$) or the complex numbers.

We have adopted a functional style of type computation, as opposed to the relational style of type derivation above, since each PVS expression has a canonical type given by the type declarations of its constants and variables.[4] With subtypes, a single expression, such as 2, can have multiple types such as `real`, `rat` (rational number), `int` (integer), `nat` (natural number), and `even`. The functional computation of a canonical type removes any possibility for nondeterminism without loss of completeness (every typeable term is assigned a canonical type, e.g., the canonical type of 2 is `real`). The soundness argument for the type system follows the definition of the τ operation and is therefore quite straightforward [OS97].

The actual PVS specification language differs from the core PVS presented above. PVS also has record types which can be captured by product types and are therefore omitted from the core language. PVS has n-ary products instead of the binary products used in the core language. In this extended abstract, we are ignoring features of PVS such as type-directed conversions, parametric theories, and recursive and inductive definitions.

If we let $\mathbf{2}$ represent a two-element set for interpreting the type `bool`, and $\mathbf{R}$ represent the set of real numbers, the semantics for the simple type system is given with respect to a universe $U = \bigcup_{i<\omega} U_i$, where

$$\begin{aligned} U_0 &= \{\mathbf{2}, \mathbf{R}\} \\ U_{i+1} &= U_i \bigcup \{X \times Y \mid X, Y \in U_i\} \bigcup \{X^Y \mid X, Y \in U_i\} \end{aligned}$$

An *assignment* for a context Γ is a list of bindings of the form $\{x_1 \leftarrow y_1\} \ldots \{x_n \leftarrow y_n\}$ that associates the type, constant, and variable declarations of Γ with subsets and elements of the universe U. A *valid* assignment is one in which the assignment of a constant or variable is an element of the assignment of its declared type. The meaning of a well-formed type A in a context Γ is given as $\mathcal{M}(\Gamma \mid \gamma)(A)$. Correspondingly, the meaning of a well-typed term a with respect to a context Γ and γ is given as $\mathcal{M}(\Gamma \mid \gamma)(a)$. The definitions and proofs of soundness can be found in the PVS semantics report [OS97].

The soundness theorem asserts that for a well-formed context Γ, and valid assignment γ, a well-formed type A, and a type-correct term a,

1. $\mathcal{M}(\Gamma \mid \gamma)(A) \in U$, and
2. $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(\tau(\Gamma)(a))$.

PVS is quite liberal about overloading so that the same symbol can be declared multiply and can also be reused as a constant, variable, type, or theory name. Names declared within parametric theories can also be used without supplying the actual parameters. The type checker uses contextual type information to resolve ambiguities arising from overloading and to determine the precise theory parameters for the resolved names. Other than resolving overloaded names and determining theory parameters, the simple type system does not pose any serious implementation challenges.

[4] This is also the style followed in the PVS Semantics Report [OS97].

2 Predicate subtyping

Predicate subtypes are perhaps the most important feature of the PVS type system. The subtype of elements of a type T satisfying the predicate p is written as $\{x : T \,|\, p(x)\}$. Here $p(x)$ can be an arbitrary PVS formula. As examples, the type of nonzero real numbers `nzreal` is given as $\{x : \texttt{real} \,|\, x \neq 0\}$, and the type of division is given as $[\texttt{real}, \texttt{nzreal} \rightarrow \texttt{real}]$. Subtypes thus allow partial functions to be expressed as total functions over a restricted domain specified as a subtype.[5]

Partial functions do have the advantage of being more expressive. The *subp* example from Cheng and Jones [CJ90] is given by

$$subp(i, j) = \text{ if } i = j \text{ then } 0 \text{ else } subp(i, j + 1) + 1 \text{ endif}$$

and is undefined if $i < j$ (when $i \geq j, subp(i, j) = i - j$). The formula

$$(subp(i, j) = i - j) \text{ OR } (subp(j, i) = j - i)$$

is perfectly meaningful in most treatments for partial functions, but since it generates unprovable obligations, it is not considered type-correct in PVS. In practice, we have yet to encounter a need for this kind of expressiveness.

Subtypes have many other uses. They can be used to specify intervals and subranges of the integers. Thus arrays can be declared as functions over an index type that is a subrange. If `below`(10) represents the subtype of natural numbers below 10, then a ten-element integer array a can be given the type $[\texttt{below}(10) \rightarrow \texttt{int}]$. Subtypes are also useful for recording properties within the type of an expression. For example, the type of the absolute value function *abs* can be given as $[\texttt{real} \rightarrow \texttt{nonnegreal}]$, where `nonnegreal` is the type of nonnegative real numbers.

Predicate subtypes correspond to subsets of the parent type. The equality relation remains the same from a type to a subtype. One might think that predicate subtypes could be translated away since any subtype constraints on quantified variables can be moved into the body of the quantification. However, this is not the case for lambda-expressions since $\lambda(x : A)\ a$ where A is a subtype, is not expressible in the system without subtypes.

Predicate subtyping on higher-order types is especially useful for introducing types corresponding to

- Injective, surjective, bijective, and order-preserving maps
- Reflexive, transitive, symmetric, anti-symmetric, partially ordered, well-ordered relations
- Monotone predicate transformers. This is at least a third-order concept.

[5] Note that in PVS, unlike in Maude or CASL, once division is declared to be of this type, no further declarations may extend its domain. Any use of division whose denominator is not known to be nonzero will generate a proof obligation when type-checked.

Predicate subtyping is orthogonal to structural subtyping used in type systems for object-oriented languages [Car97]. In particular, with predicate subtyping, subtyping on function types is not contravariant on the domain type. The function type $[A{\to}B]$ is a predicate subtype of $[A'{\to}B']$ iff B is a predicate subtype of B', and $A \equiv A'$. Since A and A' may contain predicate subtypes, type equivalence can also generate proof obligations corresponding to the equivalence on predicates.

The proof obligations generated by the PVS typechecker are called type-correctness conditions (TCCs). Subtype TCCs take into account the logical context within which a subtype proof obligation is generated. For example, the expression $x \neq y \supset (x+y)/(x-y)$ generates proof obligation $x \neq y \supset (x-y) \neq 0$ corresponding to the subtype `nonnegreal` for the denominator of the division operation. The logical context of the subtype condition is included as the antecedent to the proof obligation.

The most significant feature of subtyping in PVS is the division of typechecking into

1. Simple type correctness which is established algorithmically by the typechecker, and
2. Proof obligations corresponding to the subtype predicates that are conjectures that have to be proved within a proof system.

As a consequence, typechecking in PVS is undecidable insofar as the generated proof obligations may not always fall within a decidable fragment of the logic. This is the only source of undecidability in the PVS type system. For example, the type $\{x : \texttt{bool} \,|\, x\}$ is the subtype of booleans that are `TRUE`. Naturally, any theorem has this type and it is easy to see that typechecking with respect to such a subtype is equivalent to theorem proving in general.

Undecidability is not a serious drawback in practice. The typechecker is merely generates proof obligations without actually trying to verify them. Typical proof obligations do fall within an efficiently decidable fragment of the logic and can be discharged by simple proof strategies that rely heavily on the PVS decision procedures. The proliferation of type-correctness proof obligations is a potentially serious drawback, but is mitigated by other features of the PVS type system, particularly,

– Subsumption which ensures that when a stronger proof obligation already exists, a weaker one is never generated. For example, a TCC $x \neq y \supset x-y \neq 0$ would be subsumed by a TCC of the form $x - y \neq 0$ and would therefore be suppressed.
– Typing judgements that can cache subtype information about specific expressions. These are discussed below in greater detail.

There are two basic operations associated with typechecking in the type system with subtypes. One operation $\mu(A)$ returns the maximal supertype of a type A, and the other $\pi(A)$ returns the predicate constraints in the type A with respect to the maximal supertype $\mu(A)$. A variant $\mu_0(A)$ returns the *direct* *supertype* so that $\mu_0(\{x : T \mid a\}) = \mu_0(T)$, and otherwise, $\mu_0(T) = T$. In contrast to μ_0, $\mu([A{\rightarrow}B])$ is defined to be $[A{\rightarrow}\mu(B)]$.

Two types A and B are *compatible* iff $\mu(A)$ and $\mu(B)$ are equivalent. When typechecking an application $f\ a$ where the canonical type of f is $[A{\rightarrow}B]$ and the canonical type of a is A', we have to ensure that A and A' are compatible (which might generate type equivalence proof obligations) and discharge any proof obligations corresponding to the subtype predicates imposed by A on a. For example, the type of positive integers `posint` and the type of nonzero natural numbers `nznat` are equivalent. The compatibility proof obligations in the context Γ are represented as $(A \sim A')_\Gamma$. The type rules are given by

$$\begin{aligned}
\tau(\Gamma)(\{x : T \mid a\}) &= \texttt{TYPE},\ \text{if } \Gamma(x) \text{ is undefined},\\
&\quad\ \tau(\Gamma)(T) = \texttt{TYPE},\ \text{and } \tau(\Gamma, x : \texttt{VAR}\ T)(a) = \texttt{bool}\\
\tau(\Gamma)(f\ a) &= B,\ \text{where } \mu_0(\tau(\Gamma)(f)) = [A{\rightarrow}B],\\
&\quad\ \tau(\Gamma)(a) = A',\\
&\quad\ (A \sim A')_\Gamma,\\
&\quad\ \vdash_\Gamma \pi(A)(a)\\
\tau(\Gamma)(\texttt{p}_i\ a) &= A_i,\ \text{where } \mu_0(\tau(\Gamma)(a)) = [A_1, A_2]
\end{aligned}$$

For example, let `g: {f: [nat -> nat] | f(0) = 0}`, and `x: int`. Then

$$\begin{aligned}
\mu_0(\tau(\Gamma)(\texttt{g})) &= \texttt{[nat -> nat]},\\
\tau(\Gamma)(\texttt{x}) &= \texttt{int},\ \text{and}\\
\texttt{int} &\sim \texttt{nat},\ \text{since } \mu(\texttt{int}) = \mu(\texttt{nat}) = \texttt{number},\ \text{hence}\\
\tau(\Gamma)(\texttt{g(x)}) &= \texttt{nat},\ \text{with the proof obligation } \pi(\texttt{nat})(\texttt{x}) = \texttt{(x >= 0)}
\end{aligned}$$

The type rules with subtyping are quite a bit more complicated than those of the simple type system. The implementation of these rules within the PVS typechecker has to cope with the interaction between subtyping and name resolution since there is no longer an exact match between the domain type of a function and its argument type.

For the interpretation of subtyping, the semantic universe has to be expanded to include subsets.

$$\begin{aligned}
U_0 &= \{\mathbf{2}, \mathbf{R}\}\\
U_{i+1} &= U_i \bigcup \{X \times Y \mid X, Y \in U_i\} \bigcup \{X^Y \mid X, Y \in U_i\} \bigcup \bigcup_{X \in U_i} \wp(X)
\end{aligned}$$

The semantics of a predicate subtype is given by the definition

$$\begin{aligned}
&\mathcal{M}(\Gamma \mid \gamma)(\{x : T \mid a\})\\
&= \{y \in \mathcal{M}(\Gamma \mid \gamma)(T) \mid \mathcal{M}(\Gamma, x : \texttt{VAR}\ T \mid \gamma\{x \leftarrow y\})(a) = \mathbf{1}\}.
\end{aligned}$$

Subtyping is one of several sources of proof obligations in PVS. Other sources of proof obligations include

1. Recursive functions, corresponding to termination.
2. Parametric theory instances, corresponding to the *assumptions* in the theory about its parameters.
3. Constant definitions, since the declared type must be shown to be nonempty. This check is not strictly necessary since such a declaration corresponds to an inconsistent axiom. However, the check is there to prevent inconsistencies from being introduced through constant declarations.
4. Inductive relation definitions, since these must be defined as fixed points of *monotone* predicate transformers.

In general, proof obligations are used in PVS to implement complete, or relatively complete, semantic checks instead of incomplete syntactic checks on the well-formedness of PVS specifications.

3 Dependent Typing

The combination of dependent typing with predicate subtyping is extremely powerful and can be used to capture the relationship between the output and the input of a function. This allows the specification of an operation to be captured within the type system. The type $\texttt{below}(n)$ is actually a dependent type and is declared as

$$\texttt{below}(n) : \mathsf{TYPE} = \{s : \texttt{nat} \mid s < n\}.$$

The definition of binomial coefficients $\binom{n}{k}$ serves as a good illustration of dependent typing.

First, the factorial operation is defined recursively. Predicate subtyping is used to note that the result of $\texttt{factorial}(n)$ is always a positive integer.

```
n: VAR nat

factorial(n):
    RECURSIVE posnat =
    (IF n > 0 THEN n * factorial(n - 1)
              ELSE 1 ENDIF)
    MEASURE n
```

Then $\binom{n}{k}$ given by `chooses0(n, k)` is computed using the `factorial` operation.

```
chooses0(n, (k : upto(n))) : rat =
    factorial(n)/(factorial(k) * factorial(n-k))
```

In the definition of `chooses0`, the domain type of the operation is a dependent tuple type where the type of the second component `upto(n)` depends on the first component `n`, where `upto(n)` is defined as {`s: nat | s <= n`}. The predicate subtyping on the second argument is identical to the informal restriction given in textbook definitions [Lev90].

The type of `chooses0(n, (k : upto(n)))` has been given as `rat` instead of the more accurate `posnat`. This is because it is necessary to establish that the right-hand side of the definition is a positive integral quantity. This nontrivial proof obligation is typically overlooked in textbook presentations. In the PVS development, the definition of `chooses0` is used to prove the basic recurrence $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$, for $0 \leq k < n$. This is stated below as the lemma `chooses0_recurrence`.

```
chooses0_recurrence: LEMMA
  (FORALL (k:upto(n)):
    chooses0(n, k) =
     (IF (k = 0 OR n = k) THEN 1
      ELSE chooses0(n-1, k) + chooses0(n-1, k-1)
      ENDIF))
```

The above recurrence can be used to show that the definition of $\binom{n}{k}$ always computes a positive integral quantity.

```
chooses(n, (k : upto(n))): posnat =
   chooses0(n, k)
```

The definition of `chooses`, when typechecked, generates a proof obligation corresponding to the claim that $\binom{n}{k}$ returns a positive integral quantity. This proof obligation is discharged using the recurrence by an interactive inductive proof.[6]

PVS admits only a very restricted form of type dependency. In a dependent type $T(n)$, the parameter n can occur only within subtype predicates in $T(n)$. This means that the structure of $T(n)$ is invariant with respect to n. All possible ways of introducing type dependencies in PVS preserve this invariant. It follows that there is no way of defining a type $T(n)$, where $T(n)$ is A^n, i.e., the n-tuple over the type A. Similarly, the D_∞ model of lambda-calculus [Bar78] is also not definable as a type since its construction involves a dependent type $T(n)$ where $T(n+1) = [T(n) \rightarrow T(n)]$.

Dependent typing adds quite a bit of complexity to the type rules. The substitution operation is needed in the definition of the type rules. The definition of type equivalence and maximal supertype is not straightforward. The PVS formal semantics report [OS97] can be consulted for further details. The implementation of the typechecker for dependent typing is also correspondingly more difficult since it requires more contextual information and quite heavy use of substitution. We intend to investigate whether a representation of types using

[6] Note that `chooses0` could be defined as a `posnat` to begin with, but the resulting proof obligation is not trivial to prove. It was in attempting to prove this obligation that the `chooses0_recurrence` lemma was developed.

explicit substitutions might be more efficient for typechecking with dependent types.

4 Judgements

With subtyping, the same term can have more than one type. As we have already seen, the term 2 has the types `real`, `rat`, `int`, `nat`, `posnat`, `even`, and `prime`. An operation can return a result of a more refined subtype than its declared range type, if it is given arguments of a more refined domain type than its declared domain type. The arithmetic operation of multiplication is a good example here. The product of two positive numbers or two negative numbers is positive. Such subtype propagation information can be specified using a `JUDGEMENT` declaration. Typing judgements generate proof obligations corresponding to the validity of the judgement. The judgements are used by the typechecker in a proactive manner to propagate subtype information which minimizes the generation of redundant proof obligations.

There are two kinds of judgements in PVS. Typing judgements assert that a given operation propagates type information in a specific manner. For example, two simple judgements about the propagation of sign information by the addition operation are recorded below.

```
px, py:   VAR posreal
nx, ny:   VAR negreal
nnx, nny: VAR nonneg_real
nnreal_plus_posreal_is_posreal:   JUDGEMENT +(nnx, py) HAS_TYPE posreal
negreal_plus_negreal_is_negreal:  JUDGEMENT +(nx, ny) HAS_TYPE negreal
```

The first judgement asserts that the sum of a nonnegative and a positive real is a positive real. The second judgement asserts that the sum of two negative reals is negative. When the typechecker is applied to a term, say $(-2 + -5)$, it is able to conclude that the term has the type negative real number. Stronger judgements allow the typechecker to conclude that the term $(-2 + -5)$ has the type of negative integers. This, in turn, allows the typechecker to conclude that $(-2 + -5) + -3$ has the type `negreal`.

Judgements thus allow certain classes of proof obligations to be proved once and for all. The typechecker uses judgements to propagate type information from subterms to the terms in a proactive manner. The refined type information computed by the typechecker not only minimizes the number of proof obligations, it is also used by the PVS proof checker in simplification. For example, judgements facilitate the computation of sign information for arithmetic terms. Such sign information is recorded in the data structures of the decision procedures and is employed in arithmetic simplification. The PVS decision procedures are only modestly effective at nonlinear arithmetic so the statically inferred sign information comes in quite handy during simplification.

5 Abstract Datatypes

PVS, like many other specification languages, has a definition mechanism for a certain class of recursive datatypes given by constructors, accessors, and recognizers. The `list` datatype is given in terms of the constructors

- `null` with recognizer `null?` and with no accessors, and
- `cons` with recognizer `cons?` and accessors `car` and `cdr`.

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

The datatype is parametric in the element type `T`. This definition generates various PVS theories that contain the relevant datatype axioms and a number of useful operators for defining operations over datatype terms.

The predicate subtype of the datatype corresponding to the recognizer `cons?` is represented by the type expression `(cons?)`. Then the accessor `car` has the type [(`cons?`)→`T`] and the accessor `cdr` has the type [(`cons?`)→`list`].

Whenever an accessor is used in an expression, as in `car(cdr(x))`, the typechecker generates proof obligations requiring that `cons?(x)` and `cons?(cdr(x))` hold in the context of any conditions given by the context.

Predicate subtypes allow mutually recursive datatypes to be introduced using the same mechanism as recursive datatypes. For a simple example, suppose we wish to construct datatypes consisting of arithmetic expressions constructed from numbers by means of addition and branching, and boolean expressions that are equalities between arithmetic expressions. This could be expressed as

```
expr: DATATYPE
 BEGIN
  eq(t1: term, t2: term): eq?
 END expr

term: DATATYPE
 BEGIN
  num(n:int): num?
  sum(t1:term,t2:term):   sum?
  ift(e: expr, t1: term, t2: term): ift?
 END term
```

But now the induction schema for each of these datatypes relies on the other, making it difficult to work with.[7] We chose a simpler approach that relies on subtypes:

[7] This is similar to the problem of describing measures that decrease across mutually recursive function definitions.

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

In this datatype, `term` is the subtype {`x: arith | num?(x) OR sum?(x) OR ift?(x)`}, and a single induction schema is generated that simultaneously inducts over `terms` and `exprs`.

Ordered binary trees are another demonstration of the interaction of datatypes and predicate subtyping. The type of ordered binary trees can be defined as a subtype of the binary trees datatype that satisfies the ordering condition.

6 Comparisons

Lamport and Paulson [LP99] argue that types are harmful in a specification language. They acknowledge that predicate subtypes remedy some of the expressiveness limitations of type systems, but argue that subtypes are inherently complicated. Indeed, a sizable fraction of the bugs in early implementations of PVS were due to predicate subtyping in particular, and proof obligation generation, in general. However, these bugs stem largely from minor coding errors rather than foundational issues or complexities. The recently released PVS version 2.3 overcomes most of these problems is quite robust and efficient. Much of the popularity of PVS as a specification framework stems from its effective treatment of predicate subtyping. Predicate subtyping is not a trivial addition to a specification language, but the payoff in terms of expressiveness more than justifies the implementation cost.

The specification language VDM [Jon90] has a notion of *data type invariants* where types can be defined with constraints that are similar to those of predicate subtypes. Typechecking expressions with respect to types constrained with invariants generates proof obligations.[8] In VDM, such invariants are part of the type definition mechanism rather than the type system itself. Since VDM is based on a first-order logic, there is nothing corresponding to a higher-order predicate subtype. Dependent types are absent from the VDM type system. VDM treats partiality with a 3-valued logic instead of subtyping.

[8] To quote Jones [Jon90]:

> This [the concept of data type invariants] has a profound consequence for the type mechanism of the notation. In programming languages, it is normal to associate type checking with a simple compiler algorithm. The inclusion of a sub-typing mechanism which allows truth-valued functions forces the type checking here to rely on proofs.

Systems like HOL [GM93] and Isabelle/HOL [Pau94] are based on Church's simply typed higher-order logic [Chu40]. These have the advantage that the implementations are simple and reliable. PVS extends the simple type system in a number of ways, but these extensions are well supported by means of the proof automation in PVS. PVS has been compared with HOL by Gordon [Gor95] and with Isabelle by Griffioen and Huisman [GH98]. The type systems of PVS and Nuprl [CAB$^+$86] have been compared by Jackson [Jac96].

Dependent type theories were introduced as a formalization of constructive logics based on the Curry-Howard isomorphism. Constructive logics like AUTOMATH [dB80], Nuprl [CAB$^+$86], and Coq [DFH$^+$91] feature dependent typing in their type system. The dependencies in these logics are different from those in the PVS type system. In PVS, the dependencies can only affect the predicates in a type but not its structure. For example, the type $[n : nat \rightarrow A^n]$ cannot be defined in PVS. Whereas, the constructive type theories admit dependent types where the structure of the range can depend on the value of the argument. Nuprl also has a form of predicate subtyping but it does not separate typechecking into an algorithmic component and proof obligation generation: all typechecking is carried out within a proof by invoking the type rules. Coq has a fully polymorphic type system whereas PVS features only a limited degree of polymorphism through type parametricity at the theory level. Nuprl also has a hierarchy of type universes where the terms at each level are assigned types at the next level in the hierarchy. PVS on the other hand admits no reasoning at the level of types so that even type equivalence is algorithmically reduced to an ordinary proof obligation.

Algebraic specification languages [FGJM85a,Mos98] typically employ multi-sorted first-order logics. In contrast, PVS is based on a more expressive higher-order logic. In algebraic specification languages, subsorting is analogous to subtyping in PVS. However, the subsorting is not enforced so that, e.g., division by zero is allowed, and in the case of programming languages such as OBJ and Maude simply results in a runtime error. In the case of specification languages such as CASL, proofs involving partial terms tend to require definedness arguments. In principal, this is the same as dealing with PVS proof obligations, but in practice the PVS judgement mechanism greatly reduces the burden on the user.

7 Conclusions

We have argued that predicate subtypes are a fundamental and important extension to a specification language. They allow partial operations such as division to be given as total operations over a subtype. Properties of the result of an operation can be cached in the type. For example, $\texttt{mod}(a, b)$ can be defined so that b must be positive, and the result $\texttt{mod}(a, b)$ is at most b. In PVS, there are no restrictions on the predicates that can be used to construct predicate subtypes. Typechecking with predicate subtypes is undecidable in general. PVS separates typechecking in the presence of predicate subtypes into simple typechecking and proof obligation generation. An expression is not considered type-correct unless all generated proof obligations have been discharged.

Dependent typing allows the predicates in one component of a compound type to be defined in terms of the other components. With the combination of predicate subtyping and dependent typing, a substantial part of the specification of an operation can be embedded in its type.

With recursive datatypes, several problems associated with the use of multiple-constructor datatypes can be avoided through the use of predicate subtyping. Proof obligations ensure that an accessor is never improperly applied.

A substantial fragment of the PVS language is executable. An execution engine has been implemented for PVS by means of code generation from PVS to Common Lisp [Sha99]. The PVS type system ensures that the execution of every well-typed ground term is safe, i.e., the only possible runtime error occurs when some resource bound has been exhausted. Annotations derived from subtype information also yield an efficiency improvement of about 30%. For example, if the type of a PVS ground term is known to be positive and smaller than the Common Lisp `fixnum` type, a declaration may be added to the generated code that allows the compiler to omit some runtime checks. On some hardware simulation examples, the generated code executes at roughly a fifth of the speed of hand-crafted C.

In summary, PVS is an experimental effort aimed at supporting the development of expressive specifications for both human and machine consumption. Experiments with PVS reveal that subtyping is a crucial language feature that supports expressiveness, clarity, safety, and deductive automation. It merits close consideration for programming languages as well as specification languages.

References

[And86] Peter B. Andrews. *An Introduction to Logic and Type Theory: To Truth through Proof.* Academic Press, New York, NY, 1986.

[Bar78] H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland, Amsterdam, 1978.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[Car97] Luca Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997. Available at http://www.research.digital.com/SRC.

[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.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CJ90] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In Carroll Morgan and J. C. P. Woodcock, editors, *Proceedings of the Third Refinement Workshop*, pages 51–69. Springer-Verlag Workshops in Computing, 1990.

[dB80] N. G. de Bruijn. A survey of the project Automath. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, 1980.

[DFH$^+$91] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner. The COQ proof assistant user's guide: Version 5.6. Rapports Techniques 134, INRIA, Rocquencourt, France, December 1991.

[FGJM85a] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian K. Reid, editor, *12th ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[FGJM85b] M. Futatsugi, J. Goguen, J-P. Jouanaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of the 12th ACM Symposium on Principles of Programming*, 1985.

[Fre67a] G. Frege. Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought, 1967. First published 1879.

[Fre67b] G. Frege. Letter to Russell, 1967. Written 1902.

[GH98] 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.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[Gor95] Mike Gordon. Notes on PVS from a HOL perspective. Available at http://www.cl.cam.ac.uk/users/mjcg/PVS.html, August 1995.

[Jac96] Paul Jackson. Undecidable typing, abstract theories and tactics in Nuprl and PVS (tutorial). 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*, Turku, Finland, August 1996. Springer-Verlag.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.

[Lev90] William J. Leveque. *Elementary Theory of Numbers*. Dover, 1990. Originally published by Addison-Wesley, 1962.

[LP99] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):133–169, May 1999.

[Mos98] Peter D. Mosses. CASL: A guided tour of its design. In José Luiz Fiadeiro, editor, *Recent Trends in Algebraic Specification Languages*, number 1589 in Lecture Notes in Computer Science, pages 216–240. Springer Verlag, 1998.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[OS93] Sam Owre and Natarajan Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Extensively revised June 1997; Also available as NASA Contractor Report CR-97-206264.

[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.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[ROS98] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[Rus67] Bertrand Russell. Letter to Frege, 1967. Written 1902.

[Rus93] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993.

[Sha99] N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1999. Available at http://www.csl.sri.com/shankar/PVSeval.ps.gz.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, Cambridge, UK, 1988.

[vH67] Jean van Heijenoort, editor. *From Frege to Gödel*. Harvard University Press, Cambridge, MA, 1967.

[WR27] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, revised edition, 1925–1927. Three volumes. The first edition was published 1910–1913.