Subtypes for Specifications: Predicate Subtyping in PVS

John Rushby Sam Owre N. Shankar

Abstract—**A specification language used in the context of an effective theorem prover can provide novel features that enhance precision and expressiveness. In particular, typechecking for the language can exploit the services of the theorem prover. We describe a feature called "predicate subtyping" that uses this capability and illustrate its utility as mechanized in PVS.**

Keywords—**Formal methods, specification languages, type systems, subtypes, typechecking, consistency, PVS**

I. Introduction

For programming languages, type systems and their associated typecheckers are intended to ensure the absence of certain undesirable behaviors during program execution [1]. The undesired behaviors generally include untrapped errors such as adding a boolean to an integer, and may (e.g., in Java) encompass security violations. If the language is "type safe," then all programs that can exhibit these undesired behaviors will be rejected during typechecking.

Execution is not a primary concern for specification languages—indeed, they usually admit constructs, such as quantification over infinite domains or equality at higher types, that are not effectively computable—but typechecking can still serve to reject specifications that are erroneous or undesirable in some way. For example, a minimal expectation for specifications is that they should be consistent: an inconsistent specification is one from which some statement and its negation can both be derived; such a specification necessarily allows *any* property to be derived and thus fails to say anything useful at all. The first systematic type system (now known as the "Ramified Theory of Types") was developed by Russell [2] to avoid the inconsistencies in naïve set theory, and a simplified form of this system (the "Simple Theory of Types," due to Ramsey [3] and Church [4]) provides the foundation for most specification languages based on higher-order logic. If a specification uses no axioms (beyond those of the logic itself), then typechecking with respect to such a type system guarantees consistency. However, the consistency of specifications (such as $\boxed{2}$ in Section III) that include extra-logical axioms cannot be checked algorithmically in general, so the best that a typechecker can do in the presence of axioms is to guarantee "conservative extension" of the other parts of the specification (i.e., roughly speaking, that it does not introduce any new inconsistencies).

Since their presence weakens the guarantees provided by typechecking, it is desirable to limit the use of axioms and to prefer those parts of the specification language for which typechecking ensures conservative extension. Unfortunately, those parts are usually severely limited in expressiveness and convenience, often being restricted to quantifier-free (though possibly recursive) definitions that have a strongly constructive flavor; such specifications may resemble implementations rather than statements of required properties, and proofs about them may require induction rather than ordinary quantifier reasoning. Thus, a worthwhile endeavor in the design of type systems for specification languages is to increase the expressiveness and convenience of those constructions for which typechecking can guarantee conservative extension, so that the drawbacks to a definitional style are reduced and resort to axioms is needed less often.

In developing type systems for specification languages, we can consider some design choices that are not available for programming languages. In particular, a specification language is meant to be part of an environment that includes an effective theorem prover, so it is feasible to contemplate that typechecking can rely on general theorem proving, and not be restricted to the trivially decidable properties that are appropriate for programming languages.

"Predicate subtypes" are one example of the opportunities that become available when typechecking can use theorem proving.[1] Predicate subtypes can be used to check statically for violations such as division by zero or out-of-bounds array references, and can also express more sophisticated consistency requirements. Typechecking with respect to predicate subtypes is done by proof obligation (verification condition) generation.

In the following sections we will use simple examples to explain what predicate subtypes are, and to demonstrate their utility in a variety of situations. The examples illustrating the use of predicate subtypes are all drawn from the PVS specification language [6].

II. Predicate Subtypes

Types in specification languages are often interpreted as sets of values, and this leads to a natural association of subtype with subset: one type is a subtype of another if

The authors are with the Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA

This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931. This is an expanded version of an invited paper presented at the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering and Sixth European Software Engineering Conference, Zurich, Switzerland, September 1997. Springer-Verlag Lecture Notes in Computer Science, volume 1301, pages 4–19.

[1] Another is consistency checking for tabular specifications [5].

the set interpreting the first is a subset of that interpreting the second. In this treatment (found, for example, in Mizar [7]) the natural numbers are a subtype of the integers, but the subtyping relation does not characterize those integers that are natural numbers. *Predicate* subtypes provide such a tightly bound characterization by associating a predicate or property with the subtype. For example, the natural numbers are the subtype of the integers characterized by the predicate "greater than or equal to zero." Predicate subtypes can help make specifications more succinct by allowing information to be moved into the types, rather than stated repeatedly in conditional formulas. For example, instead of

```
∀(i, j:int):i ≥ 0 ∧ j ≥ 0 ⊃ i+j ≥ i
```

(where we use ⊃ for logical implication) we can say

```
∀(i, j:nat):i+j ≥ i
```

because `i ≥ 0` and `j ≥ 0` are recorded in the type `nat` for `i` and `j`.

Theorem proving can be required in typechecking some constructions involving predicate subtypes. For example, if `half` is a function that requires an `even` number (defined as one equal to twice some integer) as its argument, then the formula

```
∀(i:int):half(i+i+2) = i+1
```

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

```
∀(i:int):∃(j:int):i+i+2 = 2×j                                  1
```

Predicate subtypes seem a natural idea and often appear, in inchoate form, in informal mathematics. Similar ideas are also seen in formalized specification notations where, for example, the datatype invariants of VDM [8, Chapter 5] have much in common with predicate subtypes. However, datatype invariants are part of VDM's mechanisms for specifying operations in terms of pre- and post-conditions on a state, rather than part of the type system for its logic. Predicate subtypes are fully supported as part of a specification logic by the Nuprl [9], ABEL [10], Raise [11], Veritas [12], and PVS [6] verification systems. Predicate subtypes arose independently in these systems (in PVS, they came from its predecessor, EHDM, whence they were introduced from the ANNA notation [13] by Friedrich von Henke, who was involved in the design of both), and there are differences in their uses and mechanization. In Nuprl, for example, all typechecking relies on theorem proving, whereas in PVS there is a firm distinction between conventional typechecking (which is performed algorithmically) and the proof obligations (they are called Type Correctness Conditions, or TCCs in PVS) engendered by certain uses of predicate subtyping.

The circumstances in which proof obligations are generated, and other properties of predicate subtypes are described in the remainder of this paper. The examples use PVS notation, which is briefly introduced in the following section.

PVS and its Notation for Predicate Subtypes

PVS is a higher-order logic in which the simple theory of types is augmented by dependent types and predicate subtypes. Type constructors include functions, tuples, records, and abstract data types (freely generated recursive types) such as trees and lists. A large collection of standard theories is provided in libraries and in the PVS "prelude" (which is a built-in library). The PVS system includes an interactive theorem prover that can be customized with user-written "strategies" (similar to tactics and tacticals in LCF-style provers), and that provides rather powerful automation in the form of decision procedures (e.g., for ground equality and linear arithmetic over both integers and reals) integrated with a rewriter [14, 15].

As noted, some constructions involving predicate subtypes generate TCCs (proof obligations); these are not decidable in general as there are no constraints on the predicates used to induce subtypes. However, many of the TCCs encountered in practice do fall within a class that is decided by the automated procedures of PVS. In other cases, the user must develop suitable proofs interactively; this arrangement provides the flexibility of arbitrary type constraints without loss of automation on the decidable ones. Proof of TCCs can be postponed, but the system keeps track of all undischarged proof obligations and the affected theories and theorems are marked as incomplete.

A PVS specification is a collection of theories. Each theory takes a list of theory parameters and provides a list of declarations or definitions for variables, individual constants, types, and formulas. Types in PVS are built starting with uninterpreted types and primitive interpreted ones, such as `bool`, `int` (integer), `nat` (natural number), and various other numeric types. Record types are given as a list of label/type pairs: for example, `[# age:nat, years_employed:nat #]`. Tuples, such as `[nat, bool]`, are similar to records except that fields are accessed by the order of their appearance rather than by labels. Function types are introduced by specifying their domain and range types: for example, binary arithmetic operations such as addition and multiplication have the type `[[real, real]→real]`, which can also be written as `[real, real→real]`.

Functions (and predicates, which are simply functions with range type `bool`) can be defined using λ-notation, so that the predicate that recognizes even integers can be written as follows (it is a PVS convention that predicates have names ending in "?").[2]

```
even?:[int→bool] = λ(i:int):∃(j:int):i = 2×j
```

However, the following "applicative" form is exactly equivalent and is generally preferred.

```
even?(i:int):bool = ∃(j:int):i = 2×j
```

The strictness of the type hierarchy ensures that the principle of comprehension is sound in higher-order logic: that is,

[2]For ease of reading, the typeset rendition of PVS is used here; PVS can generate this automatically using its LaTeX-printer. PVS uses the Gnu Emacs editor as a front end and its actual input is presented in ASCII.

predicates and sets can be regarded as essentially equivalent.[3] PVS therefore also allows set notation for predicates, so that the following definition is equivalent to the previous two.

```
even?: [int→bool] = {i:int | ∃(j:int): i = 2×j}
```

Viewed as a predicate, the test that an integer x is even is written even?(x); viewed as a set it is written x ∈ even?. These are notational conveniences; semantically, the two forms are equivalent.

Predicates induce a subtype over their domain type; this subtype can be specified using set notation (overloading the previously introduced use of set notation to specify predicates), or by enclosing a predicate in parentheses. Thus, the following are all semantically equivalent, and denote the type of even integers.

```
even: TYPE = {i:int | ∃(j:int): i=2×j}
even: TYPE = (even?)
even: TYPE = (λ(i:int): ∃(j:int): i=2×j)
even: TYPE = ({i:int | ∃(j:int): i=2×j})
```

III. Discovering Errors with Predicate Subtypes

PVS makes no *a priori* assumptions about the cardinality of the sets that interpret its types: they may be empty, finite, or countably or uncountably infinite. When an uninterpreted constant is declared, however, we need to be sure that its type is not empty (otherwise we have a contradiction). This cannot be checked algorithmically when the type is a predicate subtype, so an "existence TCC" is generated that obliges the user to prove the fact.[4] Thus the constant declaration

```
c: even
```

generates the following proof obligation, which requires nonemptiness of the even type to be demonstrated.

```
c_TCC1: OBLIGATION ∃(x: even): TRUE
```

The existence TCC is a potent detector of erroneous specifications when higher (i.e., function and predicate) types are involved, as the following example illustrates.

Suppose we wish to specify a function that returns the minimum of a set of natural numbers presented as its argument. Definitional specifications for this function are likely to be rather unattractive—certainly involving a recursive definition and possibly some concrete choice about how sets are to be represented. An axiomatic specification, on the other hand, seems very straightforward: we simply state that the minimum is a member of the given set, and no larger than any other member of the set. In PVS this could be written as follows.

```
min(s: setof[nat]): nat                                          2

simple_ax: AXIOM
  ∀(s:setof[nat]): min(s) ∈ s ∧ ∀(n: nat): n ∈ s ⊃ min(s) ≤ n
```

Here, the first declaration gives the "signature" of the function, stating that it takes a set of natural numbers as its argument and returns a natural number as its value. The axiom simple_ax then formalizes the informal specification in the obvious way, and seems innocuous enough. However, as many readers will have noticed, this axiom harbors an inconsistency: it states that the function returns a member of its argument s—but what if s is empty?

How could predicate subtypes alert us to this inconsistency? Well, as noted earlier, sets and predicates are equivalent in higher-order logic, so that a set s of natural numbers is also a predicate on the natural numbers, and thereby induces the predicate subtype (s) comprising those natural numbers that satisfy (or, viewed as a set, are members of) s. Thus we can modify the signature of our min function to specify that it returns, not just a natural number, but one that is a member of the set supplied as its argument.[5]

```
min(s: setof[nat]): (s)
```

Now this declaration is asserting the existence of a function having the given signature and, in higher-order logic, functions are just constants of "higher" type. Because we have asserted the existence of a constant, we need to ensure that its type is nonempty, so PVS generates the following TCC.

```
min_TCC1: OBLIGATION ∃(x: [s: setof[nat] → (s)]): TRUE
```

Inspection, or fruitless experimentation with the theorem prover, should convince us that this TCC is unprovable and, in fact, false.[6] We are thereby led to the realization that our original specification is unsound, and the min function must not be required to return a member of the set supplied as its argument when that set is empty.

We have a choice at this point: we could either return to the original signature for the min function in $\boxed{2}$ and weaken its axiom appropriately, or we could strengthen the signature still further so that the function simply cannot be applied to empty sets. The latter choice best exploits the capabilities of predicate subtyping, so that is the one used here. The predicate that tests a set of natural numbers for nonemptiness is written nonempty?[nat] in PVS, so the type of nonempty sets of natural numbers is written (nonempty?[nat]), and the strict signature for a min function can be specified as follows.

```
min(s: (nonempty?[nat])): (s)
```

[3]All members of a set are of the same type in higher-order logic; this notion of "set" differs from that used in set theory where $\{a, \{a\}\}$, for example, is a valid set.

[4]If the constant is interpreted (e.g., c: even = 2), then the proof obligation is to show that its value satisfies the corresponding predicate (e.g., ∃ (j: int): 2=2×j).

[5]This is an example of a "dependent" type: it is dependent because the *type* of one element (here, the range of the function) depends on the *value* of another (here, the argument supplied to the function). Dependent typing is essential to derive the full utility of predicate subtyping. It is discussed in more detail in Section VII.

[6]A function type is nonempty if its range type is nonempty or its domain type is empty. Here the domain type is nonempty (be careful not to confuse emptiness of the domain *type*, setof[nat], with emptiness of the *argument* s), so we need to be sure that the range type, (s), is also nonempty—which it is not, when s is empty.

This declaration generates the following TCC

```
min_TCC1: OBLIGATION ∃(x: [s: (nonempty?[nat]) → (s)]): TRUE
```

which can be discharged by instantiating x with the choice function for nonempty types that is built-in to PVS.[7]

With its signature taken care of, we can now return to the axiom that specifies the essential property of the min function. First, notice that the first conjunct in the axiom simple_ax shown in $\boxed{2}$ is unnecessary now that this constraint is enforced in the range type of the function. Next, notice that the implication in the second conjunct can be eliminated by changing the quantification so that n ranges over only members of s, rather than over all natural numbers. This leads to the following more compact axiom.

```
min_ax: AXIOM ∀(s:(nonempty?[nat])),(n: (s)):  min(s) ≤ n
```

Satisfied that this specification is correct (as indeed it is), we might be tempted to make the "obvious" next step and define a max function dually.

```
max(s: (nonempty?[nat])): (s)
max_ax: AXIOM ∀(s:(nonempty?[nat])),(n: (s)):  max(s) ≥ n
```

This apparently small extension introduces another inconsistency: for what if the set s is infinite? Infinite sets of natural numbers have a minimum element, but not a maximum. Let us see how predicate subtypes could help us avoid this pitfall.

Using predicate subtyping, we can eliminate the axiom max_ax and add the property that it expresses to the range type of the max function as follows.

```
max(s: (nonempty?[nat])): {x: (s) | ∀(n: (s)): x ≥ n}        3
```

This causes PVS to generate the following TCC to ensure nonemptiness of the function type specified for max.

```
max_TCC1: OBLIGATION
  ∃(x1: [s: (nonempty?[nat]) →{x: (s)|∀(n: (s)): x≥n}]): TRUE
```

Observe that by moving what was formerly specified by an axiom into the specification of the range type, we are using PVS's predicate subtyping to mechanize generation of proof-obligations for the axiom satisfaction problem.

We begin the proof of this TCC by instantiating x1 with the (built-in) choice function choose, applied to the predicate {x: (s) | ∀(n: (s)): x ≥ n} that appears as the range type.

```
(INST + "λ(s:(nonempty?[nat])):
           choose({x: (s) | ∀(n: (s)): x ≥ n})")
```

PVS proof commands are given in Lisp syntax; the first term identifies the command (here "INST" for instantiate), the second generally indicates those formulas in the sequent (see below) to which the command should be applied (+ means "any formula in the consequent part of the sequent"), and any required PVS text is enclosed in quotes. The next two proof commands

```
(GRIND :IF-MATCH NIL)
(REWRITE "forall_not")
```

then reduce the TCC to the following proof goal. (s!1 and x!1 are the Skolem constants corresponding to the quantified variables s and x in the original formula).

```
[-1]    x!1 ≥ 0                                              4
[-2]    s!1(x!1)
  |-------
{1}     ∃(x: (s!1)): ∀(n: (s!1)): x ≥ n
```

This is a "sequent," which is the manner in which PVS presents the intermediate stages in a proof. In general, there is a collection of "antecedent" formulas (here two) above the sequent line (|-------), and a collection (here, only one) of "consequents" below; the sequent is true if the conjunction of formulas above the line implies the disjunction of formulas below (if there are no formulas below the line then we need a contradiction among those above). PVS proof commands transform the current sequent to one or more simpler (we hope) sequents whose truth implies the original one. The three proof commands shown earlier respectively instantiate an existentially quantified variable (INST), perform quantifier elimination, definition expansion, and invoke decision procedures (GRIND; the annotation :IF-MATCH NIL instructs the prover not to attempt instantiation of variables), and apply a rewrite rule (REWRITE; the rule concerned comes from the PVS prelude and changes a ∀ . . . ¬ . . . above the line into an ∃ . . . below the line, which makes it easier to read). Once again, inspection, or fruitless experimentation with the theorem prover, should persuade us that the goal $\boxed{4}$ is unprovable (we are asked to prove that any nonempty set of natural numbers has a largest element) and thereby reveals the flaw in our specification.

The flaw revealed in max might cause us to examine a specification for min given in the same form as $\boxed{3}$ to check that it does not have the same problem. This min specification generates a TCC that reduces to a goal similar to $\boxed{4}$ (with $\leq$ substituted for $\geq$ in the consequent) but, unlike the max case, this goal is true, and can be proved by appealing to the well-foundedness (i.e., absence of infinite descending chains) of the less-than ordering on natural numbers.

With the significance of well-foundedness now revealed to us, we might attempt to specify a generic min function: one that is defined over any type, with respect to a well-founded ordering on that type.

```
minspec[T: TYPE, <: (well_founded?[T])]: THEORY              5
BEGIN
  IMPORTING equalities[T]

  min((s: (nonempty?[T]))): {x: (s) | ∀(i: (s)): x < i ∨ x = i}
END minspec
```

This specification introduces a general min function in the context of a theory parameterized by an arbitrary (and possibly empty) type T, and a well-founded ordering < over that type. Notice how predicate subtyping is used in the formal parameter list of this theory to specify that < must be well-founded (the predicate well_founded? is defined in

[7]We need to demonstrate the existence of a function that takes a nonempty set of natural numbers as its argument and returns a member of that set as its value. Choice functions, which are discussed in Section IV, have exactly this property.

the PVS prelude). A proof obligation to check satisfaction of this requirement will be generated whenever the theory is instantiated. Observe that the specification has been adjusted a little to separate the < and = cases that were combined into ≤ for the special case of natural numbers.

Typechecking this specification results in the following TCC, requiring us to demonstrate that the function type asserted for `min` is nonempty.

```
min_TCC1: OBLIGATION                                          6
 ∃(x1: [s: (nonempty?[T])→{x: (s) | ∀(i: (s)): x < i ∨ x = i}]):
   TRUE
```

As before, we begin the proof of this TCC by instantiating it with the choice function `choose`, applied to the predicate `{x: (s) | ∀(i: (s)): x<i ∨ x=i}` that appears as the range type.

```
(INST + "λ(s:(nonempty?[T])):
      choose({x: (s) | ∀(i: (s)): x<i ∨ x=i})")
```

This discharges the original proof obligation, but `choose` requires its argument to be nonempty, so the prover generates a new TCC subgoal to establish this fact.

```
min_TCC1 (TCC):                                               7

  |-------
{1}    ∀(s: (nonempty?[T])):
            nonempty?[(s)]({x: (s) | ∀(i: (s)): x < i ∨ x = i})
```

This is asking us to demonstrate the existence of a minimal element for any nonempty set `s` (more precisely, it is asking us to demonstrate the nonemptiness of the set of all such minimal elements). Now the type specified for < requires it to be a well-founded ordering, and we can introduce this knowledge into the proof by the command `(TYPEPRED "<")`. The command `(GRIND :IF-MATCH NIL)` then produces the following simplified sequent.

```
{-1}    s!1(x!1)
{-2}    ∀(p: pred[T]):
         (∃(y: T): p(y))
            ⊃ (∃(y: (p)): (∀(x: (p)): (¬ x < y)))
{-3}    ∀(x: (s!1)): ¬ ∀(i: (s!1)): x < i ∨ x = i
  |-------
```

Here, the formula {-2} is expressing the well-foundedness of <; instantiating the variable `p` with `s!1` and giving a few more interactive commands, we arrive at the following sequent (this is one of two subgoals generated; the other is trivial).

```
[-1]    s!1(x!1)
  |-------
{1}    i!1 < y!1
{2}    y!1 < i!1
{3}    y!1 = i!1
```

For the specialized `min` function on natural numbers, the decision procedures completed the proof at this point, but here we recognize that this goal is not true in general, and we need the additional assumption that the relation < be trichotomous (i.e., one of the three consequents must hold, as they do on the natural numbers). Once again, predicate subtypes have led us to discover an error in our specification. We can exit the prover, modify the specification 5 to stipulate that the theory parameter < must be of type `well_ordered?[T]` (a well-ordering is one that is both well-founded and trichotomous) and rerun the proof of the TCC. This time we are successful.

Given the generic theory, we can recover `min` on the natural numbers by the instantiation `min[nat,<]`. Because of the subtype constraint specified for the second formal parameter to the theory, PVS generates a TCC requiring us to establish that < on the natural numbers is a well-ordering. This is easily done, but `min[nat,>]` correctly generates a false TCC (this theory instantiation is equivalent to our previous attempt to specify a `max` function on the naturals). However, the TCC for `min[{i: int | i < 0},>]` (i.e., the max function on the negative integers) is true and provable.

The examples in this section illustrate how a uniform check for nonemptiness of the type declared for a constant leads to the discovery of several quite subtle errors in the formulation of an apparently simple specification. In our experience, the same benefit accrues in larger specifications.

IV. Automating Proofs with Predicate Subtypes

A theorem prover is needed to discharge the proof obligations engendered by predicate subtyping. Conversely, however, predicate subtypes provide information that can actively assist a theorem prover. In this section we illustrate two ways in which predicate subtypes can help automate proofs, beginning with the use of very precise range types for functions.

A couple of the proofs in the previous section used the "choice function" `choose`. PVS actually has two choice functions defined in its prelude. The first, `epsilon`, is simply Hilbert's ε operator.

```
epsilons [T: NONEMPTY_TYPE]: THEORY
BEGIN
  p: VAR setof[T]

  epsilon(p): T

  epsilon_ax: AXIOM (∃x: x ∈ p) ⊃ epsilon(p) ∈ p
END epsilons
```

Given a set `p` over a nonempty type `T`, `epsilon(p)` is some member of `p`, if any such exist, otherwise it is just some value of type `T`. (The `VAR` declaration for `p` simply allows us to omit its type from the declarations where it is used; PVS formulas are implicitly universally quantified over their free variables.)

If `p` is constrained to be nonempty, then we can give the following specification for an `epsilon_alt` function, which is simply `epsilon` specialized to this situation (note that `T` need not be specified as `NONEMPTY_TYPE` in this case).

```
choice [T: TYPE]: THEORY
BEGIN
  p: VAR (nonempty?[T])

  epsilon_alt(p): T

  epsilon_alt_ax: AXIOM epsilon_alt(p) ∈ p
END choice
```

The new choice function epsilon_alt is similar to the built-in function choose, but if we return to the proof of min_TCC1 (recall $\boxed{6}$) but use epsilon_alt in place of choose, we find that in addition to the subgoal $\boxed{7}$, we are presented with the following.

```
  |-------                                              [8]
{1} ∀(s: (nonempty?[T])): ∀(i: (s)):
     epsilon_alt[(s)]({x: (s) | ∀(i: (s)): x<i ∨ x=i}) < i
      ∨ epsilon_alt[(s)]({x: (s) | ∀(i: (s)): x<i ∨ x=i}) = i
```

This subgoal requires us to prove that the value of epsilon_alt satisfies the predicate supplied as its argument; it can be discharged by appealing to epsilon_alt_ax, but the proof takes several steps and generates a further subgoal that is similar to $\boxed{7}$ (and proved in the same way). How is it that the choice function choose avoids all this work that epsilon_alt seems to require?

The explanation is found in the definition of choose.

```
p: VAR (nonempty?[T])

choose(p): (p)
```

This very economical definition uses a predicate subtype to specify the property previously stated in epsilon_alt_ax: namely, that the value of choose(p) is a member of p.[8] But because the fact is stated in a subtype and is directly bound to the range type of choose, it is immediately available to the theorem prover—which is therefore able to discharge the equivalent to $\boxed{8}$ internally.

A further use of subtypes to assist automation of proofs involves *encapsulation*. Below is the specification for the cardinality of a finite set.[9]

```
S: VAR finite_set
n: VAR nat

inj_set(S): (nonempty?[nat]) =
   {n | ∃(f: [(S)→below(n)]): injective?(f)}

Card(S): nat = min(inj_set(S))
```

Here, inj_set(S) is the set of natural numbers n for which there is an injection from S to the initial segment of natural numbers smaller than n (the predicate injective? is defined in $\boxed{16}$ in Section VII). Card(s) is then defined as the least such n.

This construction has the advantage of being definitional, and therefore demonstrably sound, but it is inconvenient to work with. Consequently, the PVS library provides numerous lemmas that are derived from the definition but are more suitable for automated reasoning. Unfortunately, however, the automated prover strategies may sometimes choose to open up the definition of Card when it would be better to rewrite with the lemmas.

[8]The full definition is actually choose(p): (p) = epsilon(p); this additionally specifies that choose(p) returns the same value as epsilon(p), which is useful in specifications that use both epsilon and choose.

[9]This technique was developed by Ricky Butler and Paul Miner of NASA Langley, and the specification is from the new_finite_sets library that was largely developed by them and is distributed with the current version of PVS.

One way to abstract away the definition of cardinality is to use predicate subtypes to encapsulate it in the type of a new cardinality function.

```
card(S): {n: nat | n = Card(S)}
```

This implicitly defines card because the range type is a singleton. The lemmas can then be stated in terms of card, which is used as the main cardinality function, and automated prover strategies can be used safely because there is no definition of card for them to open up inappropriately.

Whereas the previous section demonstrated the utility of predicate subtypes in detecting errors in specifications, the examples in this section demonstrate their utility in improving the automation of proofs. When properties are specified axiomatically, it can be quite difficult to automate selection and instantiation of the appropriate axioms during a proof (unless they have special forms, such as rewrite rules). Properties expressed as predicate subtypes on the type of a term are, however, intimately bound to that term, and it is therefore relatively easy for a theorem prover to locate and instantiate the property automatically. Predicate subtypes also provide a form of encapsulation, so that specifications can be written in a style that prevents the theorem prover from opening up certain definitions.

V. Enforcing Invariants with Predicate Subtypes

Consider a specification for a city phone book. Given a name, the phone book should return the set of phone numbers associated with that name; there should also be functions for adding, changing, and deleting phone numbers. Here is the beginning of a suitable specification in PVS, giving only the basic types, and a function for adding a phone number p to those recorded for name n in phone book B.

```
names, phone_numbers: TYPE
phone_book: TYPE = [names→setof[phone_numbers]]
B: VAR phone_book
n: VAR names
p: VAR phone_numbers

add_number(B, n, p): phone_book = B WITH [(n) := B(n)∪{p}]
...
```

Here, the WITH construction is PVS notation for function overriding: B WITH [(n) := B(n)∪{p}] is a function that has the same values as B, except that at n it has the value B(n)∪{p}.

Now suppose we wish to enforce a constraint that the sets of phone numbers associated with different names should be disjoint. We can easily do this by introducing the unused_number predicate and modifying the add_number function as follows.

```
unused_number(B, p): bool = ∀(n: names): ¬ p ∈ B(n)      [9]

add_number(B, n, p): phone_book =
  IF unused_number(B, p) THEN B WITH [(n) := B(n)∪{p}]
                         ELSE B ENDIF
```

If we had specified other functions for updating the phone book, they would need to be modified similarly.

But where in this modified specification does it say explicitly that different names must have disjoint sets of phone numbers? And how can we check that our specifications of updating functions such as add_number preserve his property? Both deficiencies are easily overcome with a predicate subtype: we simply change the type phone_book to the following.

```
phone_book: TYPE =                                          10
      {B: [ names → setof[phone_numbers]] |
            ∀(n, m: names): n ≠ m ⊃ disjoint?(B(n), B(m))}
```

This states exactly the property we require. Furthermore, typechecking the specification $\boxed{9}$ now causes the following proof obligation to be generated. Similar proof obligations would be generated for any other functions that update the phone book.

```
add_number_TCC1: OBLIGATION                                 11
    ∀(B, n, p): unused_number(B, p)
          ⊃ ∀(r, m: names): r ≠ m
              ⊃  disjoint?(B WITH [(n) := B(n)∪{p}](r),
                           B WITH [(n) := B(n)∪{p}](m))
```

This proof obligation, which is discharged by three commands to the PVS theorem prover, requires us to prove that a phone_book B (having the disjointness property), will satisfy the disjointness property after it has been updated by the add_number function.

The proof obligation in $\boxed{11}$ arises for the same reason as the one in $\boxed{1}$: a value of the parent type has been supplied where one of a subtype is required, so a proof obligation is generated to establish that the value satisfies the predicate of the subtype concerned. Here, the body of the definition given for add_number in $\boxed{9}$ has type [names → setof[phone_numbers]], which is the parent type given for phone_book in $\boxed{10}$, and so the proof obligation $\boxed{11}$ is generated to check that it satisfies the appropriate predicate.

Observe how this uniform check on the satisfaction of predicate subtypes automatically generates the proof obligations necessary to ensure that the functions on a data type (here, phone_book) preserve an invariant. In the absence of such automation, we would have to formulate the appropriate proof obligations manually (a tedious and error-prone process), or construct a proof obligation generator for this one special purpose. The following section describes how the same mechanism can alleviate difficulties caused by partial functions.

VI. Avoiding Partial Functions with Predicate Subtypes

Functions are primitive and total in higher-order logic, whereas in set theory they are constructed as sets of pairs and are generally partial. There are strong advantages in theorem proving from adopting the first approach: it allows use of congruence closure as a decision procedure for equality over uninterpreted function symbols, which is essential for effective automation [16]. On the other hand, there are functions, such as division, that seem inherently partial and cause difficulty to this approach. One way out of the difficulty is introduce some artificial value for undefined terms such as $x/0$, but this is clumsy and has to be done carefully to avoid inconsistencies. Another approach introduces “undefined” as a truth value [17]; more sophisticated approaches use “free logics” in which quantifiers range only over defined terms (e.g., Beeson’s logic of partial terms [18]; Parnas [19] and Farmer [20] have introduced logics similar to Beeson’s[10]). Both approaches have the disadvantage of using nonstandard logics, with some attendant difficulties. These problems have led some to argue that the discipline of types can be too onerous in a specification language, and that untyped set theory is a better choice [22].

Predicate subtypes offer another approach. Many partial functions become total if their domains are specified with sufficient precision; applying a function outside its domain then becomes a type error, rather than something that has to be dealt with in the logic. Predicate subtypes provide the tool necessary to specify domains with suitable precision.

For example, division is a total function if it is typed so that its second argument must be nonzero. In PVS this can be specified as follows.

```
nonzero_real: TYPE = {x: real | x ≠ 0}                      12
/: [real, nonzero_real→real]
```

Now consider the well-formedness of following formula.

```
test: THEOREM ∀(x, y:real): x ≠ y ⊃ (x-y)/(y-x) = -1        13
```

Subtraction is closed on the reals, so x-y and y-x are both reals. The second argument to the division function is required to have type nonzero_real; real is its parent type, so we have the proof obligation (y-x) ≠ 0, which is not true in general. However, the antecedent to the implication in $\boxed{13}$ will be false when x = y, rendering the theorem true independently of the value of the improperly typed application of division. This leads to the idea that the proof obligation should take account of the context in which the application occurs, and should require only that the application is well-typed in circumstances where its value matters. In this case, a suitable, and easily proved, proof obligation is the following.

```
test_TCC1: OBLIGATION ∀(x, y: real): x ≠ y ⊃ (y-x) ≠ 0
```

This is, in fact, the TCC generated by PVS from the formula $\boxed{13}$. PVS imposes a left-to-right interpretation on formulas, and generates TCCs that ensure well-formedness under the logical context accumulated in that order. For example, the requirements for well-formedness of an implication $P \supset Q$ are that P be well-formed, and that Q be well-formed under the assumption that P is true; if-then-else is treated as two implications, and the rules for disjunctions $P \vee Q$ and conjunctions $P \wedge Q$ are similar to that for implication, except that for disjunctions Q must be shown well-formed under the assumption that P is

[10]Farmer’s logic is used in the IMPS system [21]. IMPS generates proof obligations to ensure definedness during proofs that are similar to PVS’s TCCs. However, because the properties required to discharge these are not bound to the types, many similar proof obligations can arise repeatedly during a single proof; IMPS mitigates this problem using caching.

false. Thus, PVS generates the same TCC as above when the formula in $\boxed{13}$ is reformulated as follows.

```
test: THEOREM ∀(x, y:real): x = y ∨ (x-y)/(y-x) = -1
```

However, the accumulation of context in left-to-right order (which is sound, but conservative) causes PVS to generate the unprovable TCC `(y-x)` $\neq$ `0` for the following, logically equivalent, reformulation.

```
test: THEOREM ∀(x, y:real): (x-y)/(y-x) = -1 ∨ x = y            14
```

Another example of a partial function is the *subp* "challenge" from Cheng and Jones [23]. This function on integers is given by

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

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

As described in an earlier paper [6, Section III], this challenge is easily handled in PVS using dependent predicate subtyping to require that the second argument is no greater than the first. The function is then specified as follows.

```
subp((i:int), (j:int | j ≤ i)): RECURSIVE nat =
  IF i = j THEN 0 ELSE subp(i, j+1) + 1 ENDIF
MEASURE i-j
```

This generates three TCCs: one to ensure that the recursive call satisfies the type specified for the arguments, one to ensure that `i-j` in the `MEASURE` satisfies the predicate for `nat`, and another to establish termination using this measure. All three are discharged automatically by the PVS decision procedures.

The PVS formulation of `subp` is adequate for most purposes, but the following formula (from Maharaj and Bicarregui [24]) reveals a limitation.

```
subp_lemma: FORMULA                                              15
  ∀ i, j: nat: subp(i,j) = i-j ∨ subp(j,i) = j-i
```

This formula is true in some treatments of partial functions, but generates false TCCs and is unacceptable to PVS.

We have considered using symmetric rules for TCC generation so that examples such as $\boxed{14}$ and $\boxed{15}$ can be accepted: the left side of an expression would be typechecked in the context of the right side, as well as vice-versa, and the expression would be considered type-correct if either proof obligation can be discharged. However, we decided not to adopt this treatment for reasons of simplicity and efficiency. Since most specifications are written to be read from left to right (for the convenience of human readers), the conservatism of the left-to-right interpretation is seldom a problem in practice. (An exception is when PVS specifications are mechanically generated from some other representation.)

Predicate subtypes also yield an elegant treatment of recursive datatypes. The `stack` datatype, for example, can be specified as consisting of a constructor `empty` and another constructor `push` with accessors `top` and `pop`. In PVS, this is specified concisely as follows.

[10]The traditional notation for the second bound variable is `(j: {j: int | j ≤ i})`; PVS also allows the less redundant form used here.

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

Each constructor defines a disjoint subtype of the datatype so that the `top` and `pop` operations are total on `nonempty?` stacks and it is type-incorrect to apply them to possibly `empty?` stacks. Thus, the following definition

```
doublepop(s: (nonempty?)): stack = pop(pop(s))
```

correctly generates the following unprovable and untrue TCC.

```
doublepop_TCC1: OBLIGATION
      ∀ (s: (nonempty?[T])): nonempty?[T](pop[T](s))
```

In our experience, use of predicate subtypes to render functions total is not onerous, and contributes clarity and precision to a specification; it also provides potent error detection. As another illustration of the latter, the "domain checking" for Z specifications provided by the Z/EVES system [25] has reportedly found errors in every Z specification examined in this way [26]. (Domain checking is similar to the use of predicate subtypes described in this section, but lacks the more general benefits of predicate subtyping.)

VII. Dependent Types and Higher-Order Subtypes

Predicate subtypes are useful for defining very refined type dependencies through dependent typing. We have already seen a few examples of dependent typing, such as some of the treatments of `min` in Section III and `subp` in the previous section. Here, we illustrate its use to constrain the fields of records to "reasonable" values, and to ensure a natural treatment of equality.

Dependent typing can be used to constrain the type of one field in a record according to the value of another field, as in the following example.

```
employee_record: TYPE =
                [# age: nat, years_employed: upto(age) #]
```

This record declaration constrains the `years_employed` field to take values that are bounded above by the value of the `age` field (`upto(n)` is the type `{i: nat | i ≤ n}`). This type would thus rule out a record such as the following.

```
Jones: employee_record = (# age := 30, years_employed := 40 #)
```

The utility of the combination of dependent typing and predicate subtyping can be further illustrated by an example due to Carl Witty: the "implementation" of stacks as a record consisting of a `size` field and an array of `elements` of some type `T`. A simple formalization of this is the following record type.

```
stack_imp: TYPE = [# size: nat, elements: [nat→T] #]
```

The problem with this implementation is that two stacks that have the same `size`, say n, and agree on the first n values in the `elements` array, can still be unequal by disagreeing on irrelevant array values (i.e., those beyond `size`).[11]

[11]Note that equality on functions is extensional: two functions f and g of type `[D→R]` are equal iff for all x in `D`, $f(x) = g(x)$.

This makes it awkward to formulate correctly even such simple theorems as `pop(push(x, stack)) = stack`. The problem can be solved using dependent typing to reformulate the specification as follows.

```
stack_imp: TYPE = [# size: nat, elements: [below(size)→T] #]
```

With this refined typing, two stacks are equal when they have the same size and agree on the stack elements. Finite sequences are defined similarly in the PVS prelude.

PVS is a higher-order logic, meaning that functions can be applied to functions, and quantification can range over function types. Consequently, predicates can be defined over functions, and induce corresponding subtypes. For example, the following theory from the PVS prelude defines the predicates `injective?`, `surjective?` and `bijective?` over functions from `D` to `R`.[12]

```
functions [D, R: TYPE]: THEORY                                    16
 BEGIN
  f: VAR [D→R]
  x, x1, x2: VAR D
  y: VAR R
  injective?(f):  bool = ∀ x1, x2: f(x1)=(x2) ⊃ x1=x2
  surjective?(f): bool = ∀ y: ∃ x: f(x) = y
  bijective?(f):  bool = injective?(f) ∧ surjective?(f)
```

These induce corresponding subtypes, allowing declarations such as the following.

```
int2nat: (bijective?[int, nat]) =
      λ(i:int): IF i>0 THEN 2*i-1 ELSE -2*i ENDIF
```

By the standard mechanisms, this generates a TCC requiring a demonstration that the function `int2nat` is indeed a bijection between the integers and the naturals.

The combination of predicate subtyping, dependent typing, and higher-order types and subtypes is a powerful one. Higher-order subtypes can be used to introduce types such as order-preserving or order-inverting maps, and monotone predicate transformers. A drawback to some uses of predicate subtypes, however, is that large numbers of TCCs may be generated. In the next section we describe how this drawback can be overcome.

VIII. Judgements

The examples given in the preceding three sections illustrate the proof obligations that are generated when a term of a given type is provided where a subtype is expected. This can lead to a proliferation of many similar proof obligations. One way the PVS typechecker controls this proliferation is to check whether a new TCC is subsumed by earlier ones in the same theory: the TCC is suppressed if it is so subsumed. Although this does remove some duplicates, it is still possible to generate TCCs that differ only in some irrelevant contextual formulas.

An effective way to minimize—and often eliminate—the generation of trivially different TCCs for a given expression is for the user to state the needed proof obligation once and for all, and in its strongest form. In PVS, this is accomplished using *judgements.* The simplest form of judgement states that a given constant has a specific type. For example, we could give the judgement declaration

```
JUDGEMENT 2 HAS_TYPE even                                        17
```

This generates an immediate TCC to show that `2` is indeed even, but the typechecker can then make use of this fact to avoid generating similar TCCs in any context where the judgement is visible.

Judgements can also assert subtype constraints on the value returned by a function in terms of those on its arguments. Suppose we have the following formula declaration.

```
h: FORMULA ∀ (e: even): half(e+2) = e/2 + 1                      18
```

Recall that `half` requires an even argument. We would like the typechecker to recognize that the result of adding two even numbers is again even; this can be accomplished with the following judgement declaration.[13]

```
JUDGEMENT + HAS_TYPE [even, even → even]                         19
```

Such a judgement over a function type is interpreted as a *closure condition* equivalent to the following formula.

```
∀(e1, e2: even): even?(e1 + e2)
```

This is, in fact, the TCC generated by this judgement declaration.[14] The combination of the judgements $\boxed{17}$ and $\boxed{19}$ allows the PVS typechecker to determine immediately that the application of `half` in $\boxed{18}$ is well-typed.

The final kind of judgement informs the typechecker of subtype relations that are not explicit in their constructions. For example, the types `nonzero_real` and `nonzero_rational` are defined as follows in the PVS prelude.

```
nonzero_real: NONEMPTY_TYPE = {r: real | r ≠ 0}
nonzero_rational: NONEMPTY_TYPE = {r: rational | r ≠ 0}
```

From this the typechecker can deduce the subtype relationship between `nonzero_rational` and `rational`, and hence also the type `real` (since `rational` is defined as a subtype of `real`), but it cannot deduce a subtyping relation between `nonzero_rational` and `nonzero_real`. With division typed as in $\boxed{12}$ in Section VI, the following formula generates a TCC to show that `q` is nonzero.

```
div_lt_1: FORMULA ∃ (q: nonzero_rational): 1/q = 2
```

Such proof obligations can be avoided if the desired subtype relation is stated explicitly using following judgement.

```
JUDGEMENT nonzero_rational SUBTYPE_OF nonzero_real
```

As with other judgement declarations, this immediately generates the necessary TCC and enlarges the collection of facts known to the typechecker, thereby reducing the number of TCCs generated subsequently.

[12]Since a tuple type may be supplied for `D`, this theory is fully general and can be instantiated for functions of any arity.

[13]This, along with a large number of similar judgements, is in the PVS prelude.

[14]A future version of PVS will allow such closure constraints to be stated more directly as: `JUDGEMENT +(e1, e2: even) HAS_TYPE even`

IX. Conversions

Conversions are functions that the typechecker can insert automatically whenever there is a type mismatch. Their purpose is to provide increased syntactic convenience in situations involving subtyping of higher types, but we introduce them by means of a simpler example.

```
c: [int→bool]
CONVERSION c
two: FORMULA 2
```

Here, since formulas must be of type boolean, the typechecker automatically invokes the conversion and changes the formula to `c(2)`. This is done internally, and is only visible to the user on explicit command and in the proof checker. (To avoid confusion, the typechecker warns the user if there is ever more than one applicable conversion.)

This simple kind of conversion has nothing to do with subtypes, but standard conversions `restrict` and `extend` do play an important role in handling subtyping on function types in PVS. The rule for subtyping of function types is straightforward for the range type, so that [D→R_1] is a subtype of [D→R_2] iff R_1 is a subtype of R_2. However, the treatment of domains is less obvious. The *covariant* approach regards [D_1→R] as a subtype of [D_2→R] iff D_1 is a subtype of D_2; conversely, the *contravariant* approach regards [D_1→R] as a subtype of [D_2→R] iff D_2 is a subtype of D_1. PVS makes a more restricted choice: the domains of two functions must be *equal* for them to have a subtyping relation (which is then determined by their range types). This choice keeps the semantics simple (see Section X), but prohibits some natural constructions. Consider the following, for example.

```
g: [int→int]
F: [[nat→int]→bool]
F_app: FORMULA F(g)
```

As this stands, `F_app` is not type-correct, because a function of type `[int→int]` is supplied where one of type `[nat→int]` is required, and PVS requires equality on domain types before a subtyping relation can be considered.

However, it is clear that `g` naturally induces a function from `nat` to `int` by simply restricting its domain. Such a domain restriction is achieved by the `restrict` conversion that is defined in the PVS prelude as follows.

```
restrict [T: TYPE, S: TYPE FROM T, R: TYPE]: THEORY
BEGIN
  f: VAR [T→R]
  s: VAR S
  restrict(f)(s): R = f(s)
  CONVERSION restrict
END restrict
```

The construction `S: TYPE FROM T` specifies that the actual parameter supplied for `S` must be a subtype of the one supplied for `T`. The specification states that `restrict(f)` is a function from `S` to `R` whose values agree with `f` (which is defined on the larger domain `T`). Using this approach, a type correct version of `F_app` can be written as `F(restrict[int,nat,int](g))`. This is, of course, inconvenient to read and write, so `restrict` is specified as a conversion, which allows the PVS typechecker to insert it automatically when needed, thereby providing much of the convenience of contravariant function subtyping in this circumstance.

It is not so obvious how to expand the domain of a function in the general case, so this approach does not work so automatically in the other direction. It does, however, work well for the important special case of sets (or, equivalently, predicates): a set on some type `S` can be extended naturally to one on a supertype `T` by assuming that the members of the type-extended set are just those of the original set. Thus, if `extend(s)` is the type-extended version of the original set `s`, we have `extend(s)(x) = s(x)` if `x` is in the subtype `S`, and `extend(s)(x) = false` otherwise. We can say that `false` is the "default" value for the type-extended function. Building on this idea, we arrive at the following specification for a general type-extension function.

```
extend [T: TYPE, S: TYPE FROM T, R: TYPE, d: R]: THEORY
BEGIN
  f: VAR [S→R]
  t: VAR T
  extend(f)(t): R = IF S_pred(t) THEN f(t) ELSE d ENDIF
END extend
```

The function `extend(f)` has type `[T→R]` and is constructed from the function `f` of type `[S→R]` (where `S` is a subtype of `T`) by supplying the default value `d` whenever its argument is not in `S` (`S_pred` is the *recognizer* predicate for `S`). Because of the need to supply the default `d`, this construction cannot be applied automatically as a conversion. However, as noted above, `false` is a natural default for functions with range type `bool` (i.e., sets and predicates), and the following theory establishes the corresponding conversion.

```
extend_bool [T: TYPE, S: TYPE FROM T]: THEORY
BEGIN
  CONVERSION extend[T, S, bool, false]
END extend_bool
```

In the presence of this conversion, the type-incorrect formula `B_app` in the following specification

```
b: [nat→bool]
B: [[int→bool]→bool]
B_app: FORMULA B(b)
```

is automatically modified by the typechecker to become `B(extend[int,nat,bool,false](b))`.

These examples illustrate the utility of conversions in bringing some of the convenience of contravariant function subtyping to the more restricted type system of PVS. Conversions are also useful (for example, in semantic encodings of temporal logics) in "lifting" operations to apply pointwise to sequences over their argument types.

X. Comparison with Subtypes in Programming Languages

We know of no programming language that provides predicate subtypes, although the annotations provided for "extended static checking" (proving the absence of runtime errors such as array bound violations) [27, 28] have some similarities. Bringing the benefits of predicate subtyping to programming languages seems a worthwhile research endeavor that might generalize the benefits of extended static checking, while also providing information that could be useful to an optimizing compiler.

Subtypes of a different, "structural," kind are sometimes used in type systems for programming languages to account for issues arising in object-oriented programs [1]. In particular, a record type `A` that contains fields in addition to those of a record type `B` is regarded as a subtype of `B`. The intuition behind this kind of subtyping is rather different than the "subtypes as subsets" intuition. Here, the idea is that a subtype is an elaboration of a type, so that anywhere a value of a certain type is required, it should be acceptable to supply a value of a subtype of that type (e.g., a function that requires "points" should find a "colored point" acceptable). Structural subtypes are characterized by having a canonical coercion from the subtype to the supertype (e.g., by dropping the extra fields from a record) so that a supertype operation can be applied by means of this coercion. Using this approach, some operations can be structural subtype polymorphic—meaning that they apply uniformly to all structural subtypes of a given type. When these ideas are applied to functions, they lead to the "normal" or *covariant* subtyping on range types, but *contravariant* subtyping on domain types.

We know of no specification language that provides structural subtyping, still less combines it with predicate subtyping. The difficulty is that reasoning about equality is problematical in the presence of structural record subtypes and contravariant or covariant function subtyping.

In PVS the type of the equality relation in an equation `x = y` can be determined simply by considering the types of `x` and `y` (it is equality on their least common supertype). For example, in the expression `x/y` × `y = x`, where `x` and `y` are natural numbers, it is equality on rationals (because the division operator coerces the left hand expression to be rational). This means that if `x = y` and `y = z` are type-correct, then so is `x = z`, and it is true if the other two are. Now consider contravariant subtyping on function domains. This allows `abs = id[nat]`, where `abs: [int→nat]` is the absolute value function on the integers and `id[T]` is the identity function on type `T` (here, the naturals). This follows by promoting `abs` to its contravariant supertype `[nat→nat]` and taking equality on that type. It also allows `id[nat] = id[int]` by the same reasoning, and transitivity of equality might lead us to conclude `abs = id[int]`. But there is no reason to invoke contravariant subtyping in this final equation, since both functions have the same domain (we do need to use covariant subtyping on the range), so the equality is on the type `[int→int]`, and the equation is false. (The equation is true if equality is interpreted on `[nat→nat]` but, as noted, there is no reason to assign this type on the basis of the arguments appearing in the expression.)

Because of this difficulty in contravariant subtyping on function domains,[15] and in order to allow equals to be freely substituted, we have chosen to allow function subtyping in PVS only when the domains are equal. PVS does extend subtyping covariantly over the range types of functions (e.g., `[nat → nat]` is a subtype of `[nat → int]`) and over the positive parameters to abstract data types (e.g., list of `nat` is a subtype of list of `int`), since these cases do not present problems. We consider predicate subtyping to be a basic element of a specification language—whereas structural subtyping and subtyping on function domains are largely syntactic conveniences that we prefer to handle by mechanisms such as conversions (see also [29]), rather than incorporate them into the type system. Nonetheless, combining some structural subtyping (e.g., for records) with predicate subtyping is an interesting topic for research.

[15]Covariant subtyping on domains presents difficulties, too: it seems to require partial functions.

XI. Conclusion

We have illustrated a few circumstances where predicate subtypes contribute to the clarity and precision of a specification, to the identification of errors, and to the automation provided in analysis of specifications and in theorem proving. There are many other circumstances where predicate subtypes provide benefit, and they have been used to excellent effect by several users of PVS (see, for example, the PVS bibliography [30] and the links from its Web page). We hope the examples we have presented do convey the value of predicate subtyping in specification languages, and suggest their possible utility in programming languages.

Acknowledgments

Paul Jackson provided many suggestions that have improved the presentation.

References

Papers by SRI authors are generally available from `http://www.csl.sri.com/fm.html`.

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

[2] Bertrand Russell, "Mathematical logic as based on the theory of types," in *From Frege to Gödel*, Jean van Heijenoort, Ed., pp. 150–182. Harvard University Press, Cambridge, MA, 1967, First published 1908.

[3] F. P. Ramsey, "The foundations of mathematics," in *Philosophical Papers of F. P. Ramsey*, D. H. Mellor, Ed., chapter 8, pp. 164–224. Cambridge University Press, Cambridge, UK, 1990, Originally published in *Proceedings of the London Mathematical Society*, 25, pp. 338–384, 1925.

[4] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.

[5] Sam Owre, John Rushby, and N. Shankar, "Integration in PVS: Tables, types, and model checking," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, Ed Brinksma, Ed., Enschede, The Netherlands, Apr. 1997, vol. 1217 of *Lecture Notes in Computer Science*, pp. 366–383, Springer-Verlag.

[6] 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*, vol. 21, no. 2, pp. 107–125, Feb. 1995.

[7] Piotr Rudnicki, "An overview of the MIZAR project," in *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, Båstad, Sweden, June 1992, pp. 311–330, Available at `http://web.cs.ualberta.ca/~piotr/Mizar/MizarOverview.ps`.

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

[9] 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.

[10] Ole-Johan Dahl and Olaf Owe, "On the use of subtypes in ABEL (revised version)," Tech. Rep. 206, Department of Informatics, University of Oslo, Oct. 1995.

[11] The RAISE Language Group, *The RAISE Specification Language*, BCS Practioner Series. Prentice-Hall International, Hemel Hempstead, UK, 1992.

[12] F. Keith Hanna, Neil Daeche, and Mark Longley, "Specification and verification using dependent types," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 949–964, Sept. 1989.

[13] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe, *ANNA: A Language for Annotating Ada Programs*, vol. 260 of *Lecture Notes in Computer Science*, Springer-Verlag, 1987.

[14] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas, "PVS: Combining specification, proof checking, and model checking," In Alur and Henzinger [31], pp. 411–414.

[15] John Rushby, "Automated deduction and formal methods," In Alur and Henzinger [31], pp. 169–183.

[16] David Cyrluk, Patrick Lincoln, and N. Shankar, "On Shostak's decision procedure for combinations of theories," in *Automated Deduction—CADE-13*, M. A. McRobbie and J. K. Slaney, Eds., New Brunswick, NJ, July/Aug. 1996, vol. 1104 of *Lecture Notes in Artificial Intelligence*, pp. 463–477, Springer-Verlag.

[17] H. Barringer, J. H. Cheng, and C. B. Jones, "A logic covering undefinedness in program proofs," *Acta Informatica*, vol. 21, pp. 251–269, 1984.

[18] Michael J. Beeson, *Foundations of Constructive Mathematics*, Ergebnisse der Mathematik und ihrer Grenzgebiete; 3. Folge · Band 6. Springer-Verlag, 1985.

[19] David Lorge Parnas, "Predicate logic for software engineering," *IEEE Transactions on Software Engineering*, vol. 19, no. 9, pp. 856–862, Sept. 1993.

[20] William M. Farmer, "A partial functions version of Church's simple theory of types," *Journal of Symbolic Logic*, vol. 55, no. 3, pp. 1269–1291, Sept. 1990.

[21] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer, "IMPS: An interactive mathematical proof system," *Journal of Automated Reasoning*, vol. 11, no. 2, pp. 213–248, Oct. 1993.

[22] Leslie Lamport and Lawrence C. Paulson, "Should your specification language be typed?," SRC Research Report 147, Digital Systems Research Center, Palo Alto, CA, May 1997. Available at `http://www.research.digital.com/SRC`.

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

[24] Savi Maharaj and Juan Bicarregui, "On the verification of VDM specification and refinement with PVS," in *12th IEEE International Conference on Automated Software Engineering: ASE '97*, Incline Village, NV, Nov. 1997, IEEE Computer Society, pp. 280–289.

[25] Mark Saaltink, "The Z/EVES system," in *ZUM '97: The Z Formal Specification Notation; 10th International Conference of Z Users*, Reading, UK, Apr. 1997, vol. 1212 of *Lecture Notes in Computer Science*, pp. 72–85, Springer-Verlag.

[26] Mark Saaltink, "Domain checking Z specifications," in *LFM' 97: Fourth NASA Langley Formal Methods Workshop*, C. Michael Holloway and Kelly J. Hayhurst, Eds., Hampton, VA, Sept. 1997, NASA Langley Research Center, NASA Conference Publication 3356, pp. 185–192. Available at `http://atb-www.larc.nasa.gov/Lfm97/proceedings/`.

[27] Steven M. German, "Automating proofs of the absence of common runtime errors," in *Proceedings, 5th ACM Symposium on the Principles of Programming Languages*, Tucson, AZ, Jan. 1978, pp. 105–118.

[28] David L. Detlefs, "An overview of the Extended Static Checking system," in *First Workshop on Formal Methods in Software Practice (FMSP '96)*, San Diego, CA, Jan. 1996, Association for Computing Machinery, pp. 1–9.

[29] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov, "Inheritance as implicit coercion," *Information and Computation*, vol. 93, no. 1, pp. 172–221, July 1991.

[30] John Rushby, *PVS Bibliography*, Menlo Park, CA, Constantly updated; available at `http://www.csl.sri.com/pvs-bib.html`.

[31] Rajeev Alur and Thomas A. Henzinger, Eds., *Computer-Aided Verification, CAV '96*, vol. 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

John Rushby received B.Sc. and Ph.D. degrees in computing science from the University of Newcastle upon Tyne in 1971 and 1977, respectively. He joined the Computer Science Laboratory of SRI International in 1983, and served as its director from 1986 to 1990; he currently manages its research program in formal methods and dependable systems. Prior to joining SRI, he held academic positions at the Universities of Manchester and Newcastle upon Tyne in England.

Dr. Rushby leads the projects developing and applying PVS. His research interests center on the use of formal methods for problems in design and assurance for dependable systems. He is the author of the section on formal methods for the FAA Digital Systems Validation Handbook.

He is a member of the IEEE, the Association for Computing Machinery, the American Institute of Aeronautics and Astronautics, and the American Mathematical Society. He is an associate editor for these Transactions, and a member of the editorial board for the journal "Formal Aspects of Computing."

Sam Owre received his B.S. in mathematics from Stevens Institute of Technology in 1975, and an M.A. in mathematics from the University of California, Los Angeles in 1978. He is a Senior Software Engineer in the Computer Science Laboratory at SRI International, where for the past eight years he has devoted most of his waking hours to the development of the PVS and EHDM verification systems. Prior to that he worked in a number of AI-related research projects at Advanced Decision Systems, and before that he built yet another verification system (described in the February 1987 issue of this journal) while working at Sytek Inc. He has coauthored a number of papers on formal methods.

He is a member of the IEEE, the Association for Computing Machinery, the Association for Symbolic Logic, and the American Mathematical Society.

Natarajan Shankar received a B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras in 1980, and a Ph.D. in computer science from the University of Texas at Austin in 1986. He has been a computer scientist with the Computer Science Laboratory at SRI International since 1989. Prior to joining SRI, he was a research associate with the Stanford University Computer Science Department. His interests include formal methods, automated reasoning, metamathematics, and linear logic. He co-developed SRI's PVS specification language and verification system. His book *Metamathematics, Machines, and Gödel's Proof* was published by Cambridge University Press and is now in its second printing.

Dr. Shankar is a member of the Association for Computing Machinery, the Association for Symbolic Logic, and the IFIP Working Group 2.3 on programming methodology. He is a member of the editorial board for the journal "Formal Methods in Systems Design."