Revised 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.

Subtypes for Specifications*

John Rushby

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA

Abstract. Specification languages are best used in environments that provide effective theorem proving. Having such support available, it is feasible to contemplate forms of typechecking that can use the services of a theorem prover. This allows interesting extensions to the type systems provided for specification languages. I describe one such extension called "predicate subtyping" and illustrate its utility as mechanized in PVS.

1 Introduction

For programming languages, type systems and their associated typecheckers are intended to ensure the absence of certain undesirable behaviors during program execution [4]. 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, but typechecking can still serve to reject specifications that are erroneous or undesirable in other ways. 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 [23] to avoid the inconsistencies in naïve set theory, and a simplified form of this system (the "Simple Theory of Types," due to Ramsey [20] and Church [6]) 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. The consistency of specifications (such as $\boxed{2}$ on page 5) 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).

* 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.

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 very 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 will usually 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.[2] I am an enthusiastic user of predicate subtypes—I consider them the most useful innovation I have encountered in type systems for specification languages—and the purpose of this paper is to share my enthusiasm. I will do so using simple examples to explain what predicate subtypes are, and to demonstrate their utility in a variety of situations.

2 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 set interpreting the first is a subset of that interpreting the second. In this treatment (found, for example, in Mizar [21]) the natural numbers are a subtype of the integers, but there is nothing bound to the subtyping relation that characterizes 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 and j ≥ 0 ⊃ i+j ≥ i
```

we can say

```
∀(i, j: nat): i+j ≥ i
```

[2] Another is consistency checking for tabular specifications [17].

because `i` $\geq$ `0` and `j` $\geq$ `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 [12, 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. To my knowledge, predicate subtypes are fully supported as part of a specification logic only by the Nuprl [7] and PVS [18] verification systems. Predicate subtypes arose independently in these two systems (in PVS they came from its predecessor, EHDM, whence they were introduced from the ANNA notation [15] by Friedrich von Henke, who was involved in the design of both), and there are differences in their uses and mechanization. In Nuprl, 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 Typecheck Correctness Conditions, or TCCs) 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. Built-in types include Boolean (`bool`), and various numeric types, such as real, integer (`int`) etc. 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 [16, 22]. As noted, some constructions involving predicate subtypes generate TCCs (proof obligations); often, these can be discharged automatically using strategies provided for that purpose but, in other cases, the user must develop suitable proofs interactively. 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.

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 "?").[3]

```
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 discipline of types ensures that the principle of comprehension is sound in higher-order logic: that is, predicates and sets can be regarded as essentially equivalent.[4] 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` $\in$ `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 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})
```

3 Discovering Errors with Predicate Subtypes

PVS makes no *à 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

[3] For ease of reading, I am using the typeset rendition of PVS 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.

[4] 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.

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.[5] 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.

```
min(s: setof[nat]): (s)⁶
```

[5] 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`).

[6] 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.

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.[7] 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 I will use. 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)
```

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.[8]

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.

---

[7] A function type is nonempty if its range type is nonempty, or if both its domain and range types are 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.

[8] 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 4, have exactly this property.

```
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 and TCC-generation mechanisms 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 conclusion 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 will be a collection of "antecedent" formulas (here two) above the sequent line (`|-------`), and a collection (here, only one) of "conclusions" 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 Skolemization, 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 $\forall \ldots$ `NOT`$\ldots$ above the line into an $\exists \ldots$ 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 (it is asking us 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 conclusion) but, unlike the `max` case, this goal is true, and can be proved by appealing to the well-foundedness 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 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 $\leq$ 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 instructs the prover to expand definitions and perform other simplifications, and to Skolemize quantifiers of universal force but not to attempt to instantiate those of existential force. This produces the following simplified sequent.

```
{-1}    s!1(x!1)
{-2}    ∀(p:pred[T]):
         (∃(y:T):p(y))
             ⊃ (∃(y:(p)):(∀(x:(p)):(NOT x < y)))
{-3}    ∀(x:(s!1)):NOT ∀(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 (which it is 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 $\boxed{5}$ to stipulate that the theory parameter `<` must be of type `well_ordered?[T]` (a well-ordering is one that is 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. I have found the same benefit to accrue in larger specifications.

4 Automating Proofs with Predicate Subtypes

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` does not need to be specified as `NONEMPTY_TYPE` in this case).

```
choice [T:TYPE]:THEORY
  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 is requiring 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**.[9] 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.

Whereas the previous section demonstrated the utility of predicate subtypes in detecting errors in specifications, this example demonstrates 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 function are, however, intimately bound to that function, and it is therefore relatively easy for a theorem prover to locate and instantiate the property automatically.

5 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**.

[9] 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`.

```
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):NOT 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 ={ B:[ names → setof[phone_numbers]] |                      10
                   ∀(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.

```
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 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. This proof obligation is discharged by three commands to the PVS theorem prover.

Had there been other updating functions, similar proof obligations would have been generated automatically for them, too. This kind of proof obligation 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 FDM system of the early 1980s had such a proof-obligation generator as its core element [13]). In the following section, I show how the same mechanism can alleviate difficulties caused by partial functions.

6 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 [8]. 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 [2]; more sophisticated approaches use "free logics" in which quantifiers range only over defined terms (e.g., Beeson's logic of partial terms [3]; Parnas [19] and Farmer [10] 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 [14].

Predicate subtypes offer another approach that I find preferable to the alternatives. 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.

[10] Farmer's logic is used in the IMPS system [11]. 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.

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 }
/: [ real, nonzero_real → real ]
```

Now consider the well-formedness of following formula.

```
test:THEOREM ∀(x, y:real):x ≠ y ⊃ (x-y)/(y-x) = -1                12
```

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 $\boxed{12}$ 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 $\boxed{12}$. 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; the rules for disjunctions $P \vee Q$ and conjunctions $P \wedge Q$ are similar, except that for disjunctions Q must be shown well-formed under the assumption that P is false. Thus, PVS generates the same TCC as above when the formula in $\boxed{12}$ 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) ≠ 0` for the following, logically equivalent, reformulation.

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

Since most specifications are written to be read from left to right (for the convenience of human readers), this conservatism is seldom a problem in practice.

Another example of a partial function is the `subp` "challenge" from Cheng and Jones [5]. 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 $i \geq j$, $subp(i,j) = i - j$).

The challenge is easily handled using dependent predicate subtyping to require that the second argument is no greater than the first.

```
subp((i:int),(j:int | j ≤ i))[11]: RECURSIVE int =
  IF i = j THEN 0 ELSE subp(i, j+1) + 1 ENDIF
MEASURE i-j
```

This generates the following proof obligation from the occurrence of `j+1` in the recursive call; it is discharged automatically by the PVS decision procedures.

```
subp_TCC2: OBLIGATION
    ∀(i: int),(j: int | j ≤ i): NOT i = j ⊃ j + 1 ≤ i
```

Two other proof obligations are generated by this example: one to ensure that `i-j` in the `MEASURE` satisfies the predicate for `nat`, and another to establish termination using this measure. These are also discharged automatically by the PVS decision procedures.

In my 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. Regarding the latter, the Z/EVES system [24] provides "domain checking" for Z specifications and has reportedly found errors in every Z specification examined in this way. (Domain checking is similar to the use of predicate subtypes described in this section, but lacks the more general benefits of predicate subtyping.)

7 Comparison with Subtypes in Programming Languages

I 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) [9] 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 [4]. 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 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). When this intuition is extended to functions, it leads to the "normal" or *covariant* subtyping on range types, but *contravariant* subtyping on domain types: that is, a function type `A` is regarded as a subtype of a function type `B` if the range type of `A` is a subtype of that of `B` and if its domain type is a *super*type of that of `B`.

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

I know of no specification language that provides structural subtyping, still less combines it with predicate subtyping. There are some difficulties (e.g., preserving a simple treatment of equality) when contravariant subtyping is present, and integration of the two styles of subtyping presents an interesting research challenge. 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`), but requires equality on domain types. However, PVS also provides type "conversions" that can automatically restrict, or (less automatically) expand the domain of a function; these allow, for example, a set of `int` to be provided where a set of `nat` is expected (or vice-versa). We do expect to add some structural subtyping (e.g., for records) to PVS in future.

8 Conclusion

I 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 more circumstances where predicate subtypes provide benefit (for example, going higher-order, the injections and surjections are subtypes of the functions with the same arity; declaring a function as an injection in PVS will therefore generate the proof obligation to show that it is one-to-one), and they have been used to excellent effect by several users of PVS. I hope that the examples provided here will have persuaded you of the utility of predicate subtyping and may lead you to adopt a language that provides them, or to incorporate them in your own favorite language.

Acknowledgments

PVS and its mechanisms for predicate subtyping were developed by Sam Owre and Natarajan Shankar; I am merely an enthusiastic and grateful user of the system. This paper draws freely on their knowledge and insights. 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] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.

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

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

[4] 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`.

[5] 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.

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

[7] 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.

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

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

[10] William M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–1291, September 1990.

[11] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

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

[13] Richard A. Kemmerer. Verification assessment study final report. Technical Report C3-CR01-86, National Computer Security Center, Ft. Meade, MD, 1986. 5 Volumes (Overview, Gypsy, Affirm, FDM, and EHDM). US distribution only.

[14] 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`.

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

[16] 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 [1], pages 411–414.

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

[18] 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.

[19] David Lorge Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.

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

[21] Piotr Rudnicki. An overview of the MIZAR project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–330, Båstad, Sweden, June 1992. The complete proccedings are available at `http://www.cs.chalmers.se/pub/cs-reports/baastad.92/`; this particular paper is also available separately at `http://web.cs.ualberta.ca/~piotr/Mizar/MizarOverview.ps`.

[22] John Rushby. Automated deduction and formal methods. In Alur and Henzinger [1], pages 169–183.

[23] Bertrand Russell. Mathematical logic as based on the theory of types. In Jean van Heijenoort, editor, *From Frege to Gödel*, pages 150–182. Harvard University Press, Cambridge, MA, 1967. First published 1908.

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