Foundations of Software Engineering/European Software Engineering Conference, Zurich, Sep 97

Subtypes for Specifications

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA

Formal Methods and Calculation

- Formal methods contribute useful mental frameworks, notations, and systematic methods to the design, documentation, and analysis of computer systems
- But the primary benefit from specifically formal methods is that they allow certain questions about a design to be answered by symbolic calculation
(e.g., formal deduction, model checking)
- These symbolic calculations can be used for debugging and design exploration as well as post-hoc verification
- Comparable to the way computational fluid dynamics is used in the design of airplanes and jet engines

Corollaries

- Tools are not the most important thing about formal methods
  - They are the only important thing
  - Just like any other engineering calculations, it's tools that make formal calculations feasible and useful in practice
- Specification languages should be designed so that they support efficient calculation (i.e., deduction)
  - E.g., based on higher-order logic, not set theory
  - The topic of another talk...
- Specification languages can also be designed to exploit the efficient calculations provided by tools
  - E.g., to better detect errors in specifications
  - The topic of this talk

Errors in Formal Specifications

- Most formal specifications are full of errors
- A specification may fail to say what is intended
  - Must be examined by proving challenge theorems, "execution," and inspection
- A specification may fail to say anything at all
  - Because it is inconsistent
- Can avoid inconsistencies using definitional styles of specification that guarantee "conservative extension"
- But these are often restrictive or inappropriate (too constructive)
- So a worthwhile goal is to increase the expressiveness and convenience of the part of the specification language for which we can guarantee conservative extension

Exploiting Deduction
To Increase the Power of Typechecking

- Type systems for programming languages guarantee that certain errors will not occur during execution
- We should expect the type system for a specification language also to guarantee absence of certain kinds of errors
  - E.g., inconsistency
- Type systems for programming languages are traditionally restricted to those for which type correctness is trivially decidable
- But specification languages should be used in environments where powerful theorem proving is available, so suppose typechecking could use theorem proving. . .

Subtypes

- Subtypes can allow more concise and more precise specifications
- When types are interpreted as sets of values
  - There is a natural association of subtype with subset
  - E.g., `natural` is a subtype of `integer`
- But how do we characterize those integers that are also naturals?
- Could add an axiom

  `nat_ax: AXIOM` $\forall$`(n:nat):n` $\geq$ `0`

- But this is not tightly bound to the subtype: reduces the opportunity for automation, and may allow inconsistencies

Predicate Subtypes

- Are those where a characterizing predicate is tightly bound to subtype definitions
- For example (in the notation of PVS)

  $\texttt{nat: TYPE} = \{ \texttt{i:int} \mid \texttt{i} \geq 0 \}$

- Then we can write

  $\texttt{nat\_prop: LEMMA} \ \forall(\texttt{i, j:nat}): \texttt{i+j} \geq \texttt{i} \wedge \texttt{i+j} \geq \texttt{j}$

  And the prover can easily establish this result because the necessary information is recorded with the type for `i` and `j`

- This is concise and efficient
- Now let's see where error detection comes in

Nonemptiness Proof Obligations for Predicate Subtypes

- Subtypes may be empty, so a constant declaration

  ```
  c: nat
  ```

  Would introduce an inconsistency unless we ensure that its type is nonempty

- Generate a proof obligation called a type correctness condition (TCC) to do this

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

- Specifications are not considered typechecked until their TCCs have been discharged

Some PVS Notation

- The examples use the notation of PVS
  - A verification system freely available from SRI
- Specification language is a simply-typed higher-order logic
  - Augmented with dependent types and predicate subtypes
- Sets and predicates are equivalent in higher-order logic
- Predicates are functions of return type `bool`, written as
  `nat?(i:int):bool =` $\texttt{i} \geq 0$
  - Regarded as a predicate, membership is written `nat?(x)`
  - Regarded as a set, it is written $\texttt{x} \in \texttt{nat?}$
- A predicate in parentheses denotes the corresponding subtype
  - `(nat?)` is the same type as `nat` given earlier
- PVS has theory-level parameterization
  - `setof[nat]` is the type of sets of natural numbers

An Example: The Minimum of a Set of Naturals

- We can specify the minimum a set axiomatically as a value satisfying two properties
  - It is a member of the given set
  - It is no greater than any other member
- In PVS, this is

  $\texttt{min(s:setof[nat]): nat}$

  $\texttt{simple\_ax: AXIOM } \forall\texttt{(s:setof[nat]):min(s)} \in \texttt{s}$
  $\qquad\qquad\qquad\qquad \wedge\ \forall\texttt{(n:nat):n} \in \texttt{s} \supset \texttt{min(s)} \leq \texttt{n}$

- Unfortunately, this specification is inconsistent

The Inconsistency

- The problem is that the argument `s` to `min` could be an empty set
- But the first conjunct to `simple_ax` asserts that `min(s)` is a member of this set

Detecting the Error With Predicate Subtypes

- Using predicate subtypes, it is natural to factor the first conjunct into the return type for `min`

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

  (Observe that this is a dependent type)

- In higher-order logic, functions are just constants of "higher" type, so PVS forces us to prove that the corresponding type is not empty

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

- A (total) function type is nonempty if either
  - Its range type is nonempty, or
  - Both its domain and range types are empty

  Here, domain type is nonempty, but the range type may be

- So the TCC is false, and the inconsistency is revealed

Fixing the Specification

- Must either weaken properties of the value returned by `min`
- Or restrict its argument to be a nonempty set
- The predicate that tests for nonemptiness is `nonempty?[nat]`
- So the revised signature is

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

  And the TCC becomes

  `min_TCC: OBLIGATION` $\exists$`(x: [s: (nonempty?[nat])` $\rightarrow$ `(s)]): TRUE`

  Which is true and provable

- The second conjunct of the defining axiom can also be factored into the type

  `min(s: (nonempty?[nat])):` $\{$ `x: (s) |` $\forall$`(n: (s)): x` $\leq$ `n` $\}$

Extending the Example: from Minimum to Maximum

- It might then seem natural to define a `max` function dually

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

- This generates the following TCC

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

- To which we can apply the following PVS proof commands

```
(inst + "λ(s:(nonempty?[nat])):
           choose( {x: (s) | ∀(n: (s)): x ≥ n} )")
(grind :if-match nil)
(rewrite "forall_not")
```

- The PVS prover command grind does simplification using decision procedures and rewriting

Another Error is Revealed by Predicate Subtypes

- These proof steps produce the following goal

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

  Which is asking us to prove that any nonempty set of natural numbers has a largest member

- Not true! We are alerted to the inconsistency in our specification

- 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

Why Doesn't Minimum Have the Same Problem?

- The corresponding proof goal for `min` is

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

- Which is true, and can be proved by appealing to the well-foundedness of the naturals

```
well_founded?(<): bool =
    (∀p: (∃y: p(y))
        ⊃ (∃(y:(p)): (∀(x:(p)): (NOT x < y))))

wf_nat: AXIOM well_founded?(λ (i, j:nat): i < j)
```

This is stated in the built-in "prelude" library of PVS

A Generic Specification for Minimum

- Now we see the importance of well-foundedness, can write a generic specification for `min` over any well-founded order

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

- Notice that the constraint on `<` is stated as a predicate subtype—we will be required to prove that any actual parameter satisfies the conditions for well-foundedness

- The following nonemptiness TCC is generated

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

Discharging the TCC from the Generic Specification

- We can discharge the TCC by exhibiting the "choice function" `choose`

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

- Although this discharges the main goal, `choose` requires its argument to be nonempty (we'll see why later) so we get a subsidiary TCC proof obligation from the instantiation

```
min_TCC1 (TCC):
  |-------
{1} ∀(s: (nonempty?[T])):
       nonempty?[(s)]({x: (s) | ∀(i: (s)): x < i ∨ x = i})
```

And Another Error is Revealed by Predicate Subtyping

- The subsidiary goal looks as though it should follow by well-foundedness, so we introduce this fact as follows

```
(typepred "<")
(grind :if-match nil)
```

- And obtain the following proof goal

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

  Which is not true in general!

- We realize that well-foundedness is not enough
  - Need trichotomy as well
- Must revise specification to require that < is a well-ordering

Automating Proofs With Predicate Subtypes

- We have already seen choice functions a couple of times
- The standard one is Hilbert's ε function

```
epsilons [T: NONEMPTY_TYPE]: THEORY
BEGIN
  p: VAR setof[T]
  epsilon(p): T
  epsilon_ax: AXIOM (∃x: x ∈ p) ⊃ epsilon(p) ∈ p
END epsilons
```

- `epsilon(p)` returns a value satisfying `p` if there are any, otherwise some value of type `T`
- Notice the type `T` is required to be nonempty

Another Choice Function

- If `p` is constrained to be nonempty, can give the following specialization

```
choice [T: TYPE] : THEORY
  p: VAR (nonempty?[T])
  epsilon_alt(p): T
  epsilon_alt_ax: AXIOM epsilon_alt(p) ∈ p
END choice
```

- `epsilon_alt` is similar to the built-in `choose`, but if we use it in the proof of `min_TCC1`, we get an additional subgoal

```
  |-------
{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
```

Making Information Available in the Types

- The extra subgoal is asking us to prove that the value of `epsilon_alt` satisfies the predicate supplied as its argument
- Follows from `epsilon_alt_ax` but has quite complicated proof
- How did `choose` avoid this?
- It did so because its specification is

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

  Which records in its type the fact that `choose(p)` satisfies `p`
- It is quite hard for a prover to locate and instantiate properties stated in general axioms (unless they have special forms such as rewrite rules), but predicate subtypes bind properties to types, so the prover can locate them easily

Another Proof Obligation for Predicate Subtypes

- Suppose we introduce a new subtype for `even` integers
  `even: TYPE =` $\{$`i:int | `$\exists$`(j:int): i=2`$\times$`j`$\}$
- And a function `half` with signature
  `half: [ even` $\rightarrow$ `int ]`
- And then pose the conjecture
  `even_prop: LEMMA` $\forall$`(i: int): half(i+i+2) = i+1`
- The argument `i+i+2` to `half` has type `int`, but is required to be `even`
- However, `int` is a supertype of `even`, so we can generate a TCC to check that the value satisfies the predicate for `even`
  `even_prop_TCC1: OBLIGATION` $\forall$`(i:int):`$\exists$`(j: int): i+i+2 = 2`$\times$`j`
- PVS provides "type judgments" that allow closure properties (e.g., the sum of two `even`s is `even`) to be stated and proved once and for all

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

```
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}]
...
```

Adding an Invariant

- Suppose that different names are required to have disjoint sets of phone numbers

- Introduce `unused_number` predicate and modify `add_number` as follows

```
unused_number(B, p): bool = ∀(n: names): NOT p ∈ B(n)

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

- But where is the disjointness property stated explicitly?

- And how do we know the the modified `add_number` function preserves it?

Making the Invariant Explicit with a Predicate Subtype

- Simply change the type for `phone_book` as follows

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

- This makes the invariant explicit

- Furthermore, typechecking `add_number` generates the following TCC

```
add_number_TCC1: OBLIGATION
    ∀(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))
```

- Which requires us to prove that it really is invariant

Predicate Subtypes and Partial Functions

Many partial functions become total when their domains are specified more precisely

- For example, division can be typed as follows

  nonzero_real: TYPE = $\{$ r: real | r $\neq$ 0 $\}$

  /: [real, nonzero_real $\rightarrow$ real]

- Then typechecking

  div_prop: LEMMA $\forall$ (x, y: real):

  x $\neq$ y $\supset$ (x - y)/(y - x) = -1

- Generates the following proof obligation

  div_prop_TCC1: OBLIGATION $\forall$ (x, y: real):

  x $\neq$ y $\supset$ (y - x) $\neq$ 0

- Notice that the context of the subtype occurrence (under a left-to-right reading) appears in the TCC

- Discharged automatically by the PVS decision procedures

The `subp` "Challenge"

- Consider the function *subp* on the integers defined by

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

- This function is undefined if $j > i$
  - When $j \le i$, $\quad subp(i,j) = i - j$
- Often cited as demonstrating the need for partial functions
- But is easily handled using dependent predicate subtypes

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

subp_val: LEMMA ∀ (i,j: int): j≤i ⊃ subp(i,j) = i-j
```

The `subp` "Challenge" (continued)

- Specification generates the following TCCs

```
% Subtype TCC generated (line 6) for  i - j
  subp_TCC1: OBLIGATION ∀(i: int), (j: int | j ≤ i): i-j ≥ 0

% Subtype TCC generated (line 5) for  j + 1
  subp_TCC2: OBLIGATION
    ∀(i: int), (j: int | j ≤ i): NOT i = j ⊃ j+1 ≤ i

% Termination TCC generated (line 5) for  subp
  subp_TCC3: OBLIGATION
    ∀(i: int), (j: int | j ≤ i): NOT i = j ⊃ i - (j+1) < i-j
```

- All three proved automatically by PVS decision procedures

Higher-Order Predicate Subtypes

- It often contributes clarity and precision to a specification if functions are identified as injections, surjections, etc.
- But how to ensure a purported surjection really has the property?
- Predicate subtypes! The surjections are a subtype of the functions associated with the following predicate

```
function_props[dom, rng: TYPE]: THEORY
  surjective?(f): bool = ∀ (r: rng): ∃ (d: dom): f(d) = r
```

- So that

```
half_alt: (surjective?[even, int]) = λ (e: even): e/2
```

- Generates the following proof obligation

```
half_alt_TCC2: OBLIGATION
        surjective?[even, int](LAMBDA (e: even): e / 2)
```

Higher-Order Predicate Subtypes (continued)

The proof command (grind :if-match nil) reduces this to

```
half_alt_TCC2 :
{-1}    integer_pred(y!1)
  |-------
{1}    (∃(x: even): x / 2 = y!1)
```

Which is easily discharged

Languages with Predicate Subtypes

- The datatype invariants of VDM have some similarity to predicate subtypes (e.g., proof obligations in typechecking)
- But 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
- ACL2 has predicate "guards" on appl'ns of partial functions
- And Z/Eves does "domain checking" for Z (and has found flaws thereby in every specification checked)
- Predicate subtypes are fully supported as part of a specification logic only by Nuprl and PVS (as far as I know)
- In Nuprl, all typechecking requires theorem proving
- PVS separates the algorithmic elements from those that require TCCs

Other Approaches to Subtyping

- Some treatments of programming languages use "structural" subtypes to account for Object-Oriented features
- Intuition is different to "subtypes as subsets": value of subtype allowed anywhere one of the parent type is
- Thus adding fields to a record creates a subtype: function that operates on "points" should also operate on "colored points"
- Extension to function types leads to normal or covariant subtyping on range types
    - `[ nat → nat]` is a subtype of `[ nat → int]`
- But contravariant subtyping on domain types
    - `[ int → nat]` is a subtype of `[ nat → nat]`

Combinations

- Interesting research challenge to combine structural with predicate subtyping (contravariance complicates equality)
- PVS does extend subtyping covariantly over range types of functions
- And over the positive parameters to abstract data types
  - E.g., `list[nat]` is a subtype of list of `list[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
  - E.g., allow `setof[int]` to be provided where `setof[nat]` is expected (and vice-versa)

Conclusion

- Predicate subtypes in PVS due to Sam Owre and N. Shankar
  - Via earlier Ehdm system, where Friedrich von Henke adopted them from ANNA
  - Formal semantics available on the PVS web site
- I find them the most useful innovation I have seen in specification language design
- Many users exploit them very effectively
- I hope to have persuaded you of their value, too
- But not everybody agrees: Lamport and Paulson make a case for untyped specification languages (when used by hand)

To Learn More About PVS

- Browse general papers and technical reports at `http://www.csl.sri.com/fm.html`
  - `tse95.html` is a good overview of PVS
  - `pvs-bib.html` links to a bibliography with over 140 entries
- Detailed Information about PVS is available from `http://www.csl.sri.com/pvs.html`
  - `http://www.csl.sri.com/pvs/examples` gets you to a directory of tutorials and their supporting specification files
- You can get PVS from `ftp://ftp.csl.sri.com/pub/pvs`
  - Allegro Lisp for SunOS, Solaris, IBM AIX, or Linux
  - Need 64M memory, 100M swap space, Sparc 20 or better (Best is 64MB 200 MHz P6, costs under $3,000 in USA)