Predicate Subtyping with Predicate Sets

Joe Hurd

University of Cambridge

1. Motivation
2. Predicate Sets
3. Subtype Checking
4. Applications
5. Conclusion
6. Future Work

Motivation

Predicate subtyping allows the creation of a new subtype corresponding to an arbitrary predicate $P : \alpha \to \mathbb{B}$, where elements of the new subtype are also elements of α.

They allow (much) more information to be encoded in types, which is useful for expressing side conditions of theorems:

$$\forall\, x : \mathbb{R}^{\neq 0}.\; x/x = 1$$

The predicate subtype $\mathbb{R}^{\neq 0}$ of $\mathbb{R}$ corresponds to the predicate $\lambda\, r.\; r \neq 0$.

Predicate subtypes are powerful enough to express dependent types; and type-checking specifications using predicate subtypes allows more consistency checks to be made, thus catching errors earlier.

Motivation

However, there are some downsides: Predicate subtypes complicate the logical kernel, which has demonstrated itself in the higher number of soundness bugs in PVS relative to HOL. Also, type-checking becomes undecidable, so (potentially human) theorem-proving effort is required before a term can be accepted into the system.

In this experiment, we stick to the HOL logic, and see if we can gain the benefits of predicate subtyping by reasoning with predicate sets.

Predicate Sets

Predicate sets are an encoding of sets in higher-order logic, and all the usual set operations can be defined on them (e.g., $\in$, $\cup$ and image).

Here are some examples of predicate sets:

$$\begin{aligned}
\mathsf{nzreal} &= \{x : \mathbb{R} \mid x \neq 0\} \\
\mathsf{posreal} &= \{x : \mathbb{R} \mid 0 < x\} \\
\mathsf{nnegreal} &= \{x : \mathbb{R} \mid 0 \leq x\} \\
\mathsf{real} &= \{x : \mathbb{R} \mid \top\} \\
\forall n.\ \mathsf{lenum}\ n &= \{m : \mathbb{N} \mid m \leq n\} \\
\forall n.\ \mathsf{nlist}\ n &= \{l : \alpha^*. \mid \mathsf{length}\ l = n\} \\
\forall p.\ \mathsf{list}\ p &= \{l : \alpha^*. \mid \forall x.\ \mathsf{mem}\ x\ l \Rightarrow x \in p\}
\end{aligned}$$

We are going to reason with such sets to simulate predicate subtyping, and will call them *subtypes*.

Predicate Sets

Look again at the predicate set list:

$$\forall\, p.\ \mathsf{list}\ p = \{l : \alpha^*.\ \mid \forall\, x.\ \mathsf{mem}\ x\ l \Rightarrow x \in p\}$$

We can similarly define a predicate set for any datatype, e.g.,

$$\forall\, p, q.\ \mathsf{pair}\ p\ q = \{x : \alpha \times \beta \mid \mathsf{fst}\ x \in p \wedge \mathsf{snd}\ x \in q\}$$

We call these *subtype constructors*. Automatically making these definitions is an example of *polytypic programming*, and could be incorporated into the existing HOL datatype package.

Predicate Sets

We can define a subtype constructor for function spaces:

$$\forall\, p, q.\ p \dot{\rightarrow} q = \{f : \alpha \rightarrow \beta \mid \forall\, x \in p.\ f(x) \in q\}$$

For example:

$$(\lambda\, x.\ x^2) \in \mathsf{nzreal} \dot{\rightarrow} \mathsf{posreal}$$

A more general definition is the *dependent* subtype constructor for function spaces:

$$\forall\, p, q.\ p \stackrel{\star}{\rightarrow} q = \{f : \alpha \rightarrow \beta \mid \forall\, x \in p.\ f(x) \in q(x)\}$$

For example:

$$- \in \mathsf{num} \stackrel{\star}{\rightarrow} (\lambda\, n.\ \mathsf{lenum}\ n \dot{\rightarrow} \mathsf{lenum}\ n)$$

Subtype Checking

We say that a term t has subtype p if the HOL theorem $\vdash t \in p$ is valid. In PVS this would take the form of a type judgement (made by the type system).

We can automatically derive (some) subtypes for terms, using an algorithm similar to Milner's type inference algorithm. *Subtype rules* like the following keep track of the logical context:

$$\begin{aligned} \vdash \forall\, & c, a, b, p. \\ & (c \in \mathsf{bool}) \quad \wedge \\ & (c \Rightarrow a \in p) \quad \wedge \\ & (\neg c \Rightarrow b \in p) \quad \Rightarrow \\ & (\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b) \in p \end{aligned}$$

But how to find the common subtype p shared by a and b?

Subtype Checking

We implement a prover to find common subtypes needed by the subtype rules. This is a higher-order version of model elimination, since we need to return multiple satisfying subtypes (Prolog-style).

It needs to be higher-order because of some subtype rules, for example the one for function spaces:

$$\begin{aligned} \vdash \forall\, f, a, p, q.\ & \\ & (f \in p \xrightarrow{\star} q) \quad \wedge \\ & (a \in p) \quad \Rightarrow \\ & f\ a \in q\ a \end{aligned}$$

The prover works by translating the goal to combinatory form, following a paper of Robinson (and encouragement from John Harrison).

Subtype Checking

To be able to perform subtype checking, we also need a dictionary of *constant subtypes*

$\vdash \mathsf{inv} \in (\mathsf{nzreal} \dot{\to} \mathsf{nzreal} \cap \mathsf{posreal} \dot{\to} \mathsf{posreal} \cap \ldots)$

$\vdash \mathsf{sqrt} \in (\mathsf{nnegreal} \dot{\to} \mathsf{nnegreal} \cap \mathsf{posreal} \dot{\to} \mathsf{posreal})$

$\vdash \forall p.\ \mathsf{funpow} \in (p \dot{\to} p) \dot{\to} p \dot{\to} p$

$\vdash \forall p.\ [\,] \in (\mathsf{list}\ p \cap \mathsf{nlist}\ 0)$

$\vdash \forall p, n.$

$\quad \mathsf{cons} \in p \dot{\to} (\mathsf{list}\ p \cap \mathsf{nlist}\ n) \dot{\to} (\mathsf{list}\ p \cap \mathsf{nlist}\ (\mathsf{suc}\ n))$

$\vdash \forall f, p, q, n.$

$\quad \mathsf{map} \in (p \dot{\to} q) \dot{\to} (\mathsf{list}\ p \cap \mathsf{nlist}\ n) \dot{\to} (\mathsf{list}\ q \cap \mathsf{nlist}\ n)$

and a few *subtype judgements* for the higher-order prover to use

$\vdash \mathsf{posreal} \subset \mathsf{nzreal}$

$\vdash \forall p, q.\ p \subset q \Rightarrow \mathsf{list}\ p \subset \mathsf{list}\ q$

Subtype Checking

Consider the following (faulty) specification:

$$\vdash \ (\mathsf{inv}\ x) * x = 1$$

When the subtype checker produces an error, we can fix it like this:

$$\vdash \ x \in \mathsf{nzreal} \ \Rightarrow \ (\mathsf{inv}\ x) * x = 1$$

Or we can attempt to subvert the subtype checker in a number of ways:

$$\vdash \ \mathsf{inv}\ x \in \mathsf{nzreal} \ \Rightarrow \ (\mathsf{inv}\ x) * x = 1$$

$$\vdash \ \mathsf{inv} \in \mathsf{real} \stackrel{\cdot}{\rightarrow} \mathsf{nzreal} \ \Rightarrow \ (\mathsf{inv}\ x) * x = 1$$

$$\vdash \ \mathsf{inv} \in \mathsf{real} \stackrel{\cdot}{\rightarrow} \mathsf{real} \ \Rightarrow \ (\mathsf{inv}\ x) * x = 1$$

Subtype Checking

What went wrong?

The problem is neatly expressed in the following HOL theorem:

$$\vdash \forall f : \alpha \to \beta.\ f \in \mathcal{U}_\alpha \dot{\to} \mathcal{U}_\beta$$

This is a simple reflection of the fact that in HOL every function is total, or equivalently that *every function can be applied to any argument.* In the face of this, our attempts to restrict the domain of any function (such as inv) are ultimately futile.

Essentially the PVS logic implements a logic of partial functions, but by insisting that a type is available for every function's domain can avoid awkward questions of definedness.

Applications

We can package the subtype checker up as a HOL tactic that can solve some interesting set membership goals, for example:

$$\vdash \mathsf{map\ inv\ (cons\ } (-1) \mathsf{\ (map\ sqrt\ } [3,1])) \in \mathsf{list\ nzreal}$$

$$\vdash (\lambda\, x \in \mathsf{negreal.\ funpow\ inv\ } n\ x) \in \mathsf{negreal} \dot{\rightarrow} \mathsf{negreal}$$

We use this tactic as a condition prover in a contextual rewriter, allowing us to use the following example rewrites:

$$\vdash\ \forall\, x \in \mathsf{nzreal.}\ x/x = 1$$

$$\vdash\ \forall\, n.\ \forall\, m \in \mathsf{lenum}\ n.\ m + (n - m) = n$$

$$\vdash\ \forall\, n \in \mathsf{nznum.}\ n\ \mathsf{mod}\ n = 0$$

$$\vdash\ \forall\, s \in \mathsf{finite.}\ \forall\, f \in \mathsf{injection}\ s.\ |\mathsf{image}\ f\ s| = |s|$$

The first of these enables us to straightforwardly prove goals such as $3/3 = 5/5$.

Applications

We have applied these tools to the formalization of a body of group theory, defining subtypes such as:

$$\vdash * \in \mathsf{group} \xrightarrow{\star} (\lambda\, G.\ |G| \xrightarrow{\cdot} |G| \xrightarrow{\cdot} |G|)$$
$$\vdash e \in \mathsf{group} \xrightarrow{\star} (\lambda\, G.\ |G|)$$
$$\vdash \cdot^{-1} \in \mathsf{group} \xrightarrow{\star} (\lambda\, G.\ |G| \xrightarrow{\cdot} |G|)$$

Here $|G|$ refers to the carrier set of the group G.

Merely proving these subtype theorems and entering these into our dictionary allow us to derive group membership facts of many subterms.

We also add some subtype judgements:

$$\forall\, G \in \mathsf{group}.\ \forall\, H \in \mathsf{subgroup}\ G.\ |H| \subset |G|$$

Applications

We can then use rewrites with group membership conditions:

- $\vdash \quad \forall G \in \mathsf{group}.\, \forall g \in |G|.\, e_G *_G g = g$
- $\vdash \quad \forall G \in \mathsf{group}.\, \forall g, h \in |G|.\, (g *_G h = h) = (g = e_G)$
- $\vdash \quad \forall G \in \mathsf{group}.\, \forall H \in \mathsf{subgroup}\ G.\, e_H = e_G$
- $\vdash \quad \forall G \in \mathsf{group}.\, \forall H \in \mathsf{subgroup}\ G.\, \forall g, h \in |H|.$
  $g *_H h = g *_G h$

These allow complicated terms such as[a]

$$((h_1 *_H e_H) *_H h_2) *_G g = e_G *_G g$$

to be rewritten to

$$((h_1 *_G e_G) *_G h_2) = e_G$$

using the contextual rewriter and subtype prover in concert.

[a] where $G \in \mathsf{group}$, $H \in \mathsf{subgroup}\ G$, $g \in |G|$, $h_1, h_2 \in |H|$

Conclusions

This approach is effective at providing simple knowledge about subterms that come in useful during rewriting.

It is also useful for debugging specifications, though due to the HOL logic cannot be totally guaranteed (as it is in PVS).

When could such a guarantee be useful? Claim only when there will be no subsequent verification (since this would expose any such problem).

So our tools should be seen as an aid to formal verification, and their uses in proving conditions and debugging specifications do just that.

Future Work

- Larger case study involving arithmetic. Currently working on computational number theory, containing plenty of group theory (which is relatively easy for subtyping) and number theory (which is hard).
- Must improve the subtype infrastructure (it runs rather slowly), particularly the higher-order prover.
- Subtype inference for particular variables: this does not seem to be straightforward but would ease the burden of proving subtype theorems for every new constant.
- Mike Gordon suggested using subtypes to reformulate Hoare logic. If $[P]C[Q]$, then the program C is a function having subtype $P \dot{\rightarrow} Q$.