Predicate Subtyping with Predicate Sets

Joe Hurd*

Computer Laboratory
University of Cambridge
joe.hurd@cl.cam.ac.uk

Abstract. We show how PVS-style predicate subtyping can be simulated in HOL using predicate sets, and explain how to perform subtype checking using this model. We illustrate some applications of this to specification and verification in HOL, and also demonstrate some limits of the approach. Finally we report on the effectiveness of a subtype checker used as a condition prover in a contextual rewriter.

1 Introduction

HOL [4] and PVS [13] are both interactive theorem-provers extending Church's simple type theory [1]: in HOL with Hindley-Milner polymorphism [11]; and in PVS with parametric theories[1] and predicate subtyping [14]. In this paper we will focus on PVS predicate subtyping, and show how it can be simulated in HOL.

Predicate subtyping allows the creation of a new subtype corresponding to an arbitrary predicate, where elements of the new type are also elements of the containing type. As a simple illustration of this, the type of real division ($/$) in HOL is $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$, and in PVS is $\mathbb{R} \to \mathbb{R}^{\neq 0} \to \mathbb{R}$, where $\mathbb{R}$ is the type of real numbers and $\mathbb{R}^{\neq 0}$ is the predicate subtype of non-zero real numbers, expressed by the predicate $\lambda\, x.\ x \neq 0$.

This extension of the type system allows more information to be encoded in types, leading to benefits for specification and verification such as:

- The ability to express dependent types, for example a type for natural number subtraction that prevents the second argument from being larger than the first argument, or a type representing lists of a fixed length in order to model arrays.

* Supported by an EPSRC studentship

[1] Parametric theories allow polymorphism at the granularity of the theory (think C++ templates), whereas Hindley-Milner polymorphism operates at the granularity of the declaration (think ML functions).

– Greater use of types to express side-conditions of theorems, for example the PVS rewrite rule

$$\vdash_{\mathrm{PVS}} \forall\, x : \mathbb{R}^{\neq 0}.\ x/x = 1 \tag{1}$$

In HOL this would have to be expressed

$$\vdash \forall\, x : \mathbb{R}.\ (x \neq 0) \Rightarrow (x/x = 1) \tag{2}$$

and the extra condition must be proved each time the rule is applied.[2]

– More errors can be found in specifications during type-checking, giving greater confidence that the goal is correct before a verification is embarked upon. Mokkedem [12] has observed this to be very effective in a large network protocol verification performed in PVS. Many specifications are not verified at all, and in that case the extra confidence is especially valuable.

However, there are also some costs:

– Type-checking becomes undecidable, so (potentially human) effort must be expended to allow terms to be accepted into the system.
– Type-correctness depends on the current logical context, imposing an extra burden on term operations to keep careful track of what can be assumed at each subterm. In the case of users wishing to program their own tactics this merely steepens the learning curve; for term operations in the logical kernel faulty implementations have produced a string of soundness bugs.[3]

In the literature there are papers arguing for and against predicate subtyping. Shankar [17] gives many examples of their utility, while Lamport [9] gives an example where the costs must be paid without much benefit, in his case because predicate subtypes cannot naturally encode the desired invariant.

In this paper we describe our efforts to gain the functionality of predicate subtypes in HOL, without altering the logic in any way. Instead of creating a first-class type associated with a particular predicate P, we reason with the subset of elements that satisfy P. With this substitution, it is possible to simulate the extra reasoning power needed to automatically prove HOL side-conditions that would be expressed in PVS using predicate subtypes (such as the condition of Theorem 2 above). Using the same technology, we can also perform an analogue

[2] Analogously, typed logics such as HOL and PVS enjoy this advantage over an untyped logic such as ZF set theory, in which the theorem $\vdash \forall\, n : \mathbb{N}.\ n + 0 = n$ must be expressed $\vdash \forall\, n.\ n \in \mathbb{N} \Rightarrow (n + 0 = n)$.

[3] Shankar [17] writes "these bugs stem largely from minor coding errors rather than foundational issues or complexities". However, the extra complexity generated by predicate subtyping does play a part in this: there have been very few soundness bugs found in HOL over the years.

of predicate subtype-checking for terms, although as we shall see the HOL logic imposes certain limitations on this.

The structure of the paper is as follows: in Section 2 we lay out the details of our formalism and explain how to perform subtype-checking; Section 3 describes some tools that have been built using this technology, and reports on their effectiveness in a case study; in Section 4 we consider some fundamental logical limits of this approach; and finally in Sections 5, 6 and 7 we conclude, examine future prospects, and look at related work.

1.1 Notation

We use sans serif font to notate HOL constants, such as the function power operator funpow, the real number function inv (multiplicative inverse or reciprocal) and the list functions length and mem. This font is also used for predicate sets such as nzreal and posreal, in contrast to types (either HOL simple types or PVS predicate subtypes) which are written in mathematical font. Examples of simple types are α, β (type variables), $\alpha \times \beta$ (pairs), α^* (lists), $\mathbb{B}$ (booleans) and $\mathbb{R}$ (the real numbers); whereas $\mathbb{R}^{\neq 0}$ and $\mathbb{R}^{\geq 0}$ are predicate subtypes.

2 The Formalism

2.1 Subtypes

A predicate set is a function $P : \alpha \to \mathbb{B}$ which represents the set of elements $x : \alpha$ for which $P\ x = \top$. Note that P is parameterized by the type α, and we shall use the terminology 'α predicate set' (or just 'α set') when we wish to make this dependency explicit. Predicate sets are a standard modelling of sets in higher-order logic, and we can define polymorphic higher-order constants representing all the usual set operations such as $\in$, $\cup$ and image. In addition, for each α there exists a universe set $\mathcal{U}_\alpha = (\lambda\, x : \alpha.\ \top)$ that contains every element of type α.[4]

As examples, consider the following definitions of predicate sets that we will make use of:

$$\mathsf{nzreal} = \lambda\, x : \mathbb{R}.\ x \neq 0 \tag{3}$$

$$\mathsf{posreal} = \lambda\, x : \mathbb{R}.\ 0 < x \tag{4}$$

$$\mathsf{nnegreal} = \lambda\, x : \mathbb{R}.\ 0 \leq x \tag{5}$$

[4] Since the HOL logic specifies that all types are disjoint, so must be these universe sets.

$$\forall n : \mathbb{N}.\ \mathsf{lenum}\ n = \lambda m : \mathbb{N}.\ m \leq n \tag{6}$$

$$\forall n : \mathbb{N}.\ \mathsf{nlist}\ n = \lambda l : \alpha^*.\ \mathsf{length}\ l = n \tag{7}$$

$$\forall p : \alpha \to \mathbb{B}.\ \mathsf{list}\ p = \lambda l : \alpha^*.\ \forall x : \alpha.\ \mathsf{mem}\ x\ l \Rightarrow x \in p \tag{8}$$

$$\forall p : \alpha \to \mathbb{B}.\ \forall q : \beta \to \mathbb{B}.\ \mathsf{pair}\ p\ q = \lambda x : \alpha \times \beta.\ \mathsf{fst}\ x \in p \wedge \mathsf{snd}\ x \in q \tag{9}$$

Definitions 3–5 are straightforward, each representing the set of real numbers that are mapped to $\top$ by the predicate on the right hand side of the definition. Definitions 6–9 are all parameterized by various terms: in the case of $\mathsf{lenum}\ n$ by a natural number n so that $\mathsf{lenum}\ n = \{0, 1, \ldots, n\}$. Definitions 7–9 are polymorphic too, which of course is just another way of saying they are parameterized by types as well as terms. The set $\mathsf{nlist}\ n$ contains all α-lists having length n; the set $\mathsf{list}\ p$ contains all α-lists satisfying the condition that each member must lie in the parameter set p; and finally the set $\mathsf{pair}\ p\ q$ contains all $(\alpha \times \beta)$-pairs (x, y) where x lies in the first parameter set p and y lies in the second parameter set q.

2.2 Subtype Constructors

The definitions of list (8) and pair (9) in the previous subsection illustrated 'lifting': an α set is lifted to an α-list set using list, and an α set and a β set are lifted to a $(\alpha \times \beta)$ set using pair. We might thus call list and pair subtype constructors, and we can similarly define subtype constructors for every datatype. The automatic generation of these constuctors might be a worthwhile addition to the datatype package.

We can also define a subtype constructor for the function space $\alpha \to \beta$:

$$\forall p : \alpha \to \mathbb{B}.\ \forall q : \beta \to \mathbb{B}.\ p \mathrel{\dot{\to}} q = \lambda f : \alpha \to \beta.\ \forall x : \alpha.\ x \in p \Rightarrow f\ x \in q \tag{10}$$

The type annotations slightly obscure this definition, without them it looks like this: $\forall p, q.\ p \mathrel{\dot{\to}} q = \lambda f.\ \forall x \in p.\ f\ x \in q$.[5] Given sets p and q, f is in the set $p \mathrel{\dot{\to}} q$ iff it maps every element of p to an element of q.

We can illustrate this function subtype constructor with the following theorems that follow from the definitions so far:[6]

$$\vdash (\lambda x : \mathbb{R}.\ x^2) \in \mathsf{nzreal} \mathrel{\dot{\to}} \mathsf{nzreal} \tag{11}$$

[5] The notation $\forall x \in p.\ M\ x$ is a restricted universal [18], and expands to $\forall x.\ x \in p \Rightarrow M\ x$. There are restricted versions of all the usual HOL quantifiers.

[6] Note that $\mathrel{\dot{\to}}$ associates to the right and has tighter binding than $\in$, so $f \in p \mathrel{\dot{\to}} q \mathrel{\dot{\to}} r$ means the same as $f \in (p \mathrel{\dot{\to}} (q \mathrel{\dot{\to}} r))$. Also x^2 here means x squared.

$$\vdash (\lambda\, x : \mathbb{R}.\ x^2) \in \mathcal{U}_{\mathbb{R}} \dot{\to} \mathsf{nnegreal} \tag{12}$$

$$\vdash \forall\, f, p.\ f \in p \dot{\to} \mathcal{U}_{\mathbb{R}} \tag{13}$$

$$\vdash \forall\, f, q.\ f \in \emptyset \dot{\to} q \tag{14}$$

$$\vdash \forall\, f, p.\ f \in p \dot{\to} \emptyset \iff p = \emptyset \tag{15}$$

There is an alternative subtype constructor for function spaces, defined like so:

$$\begin{aligned} &\forall\, p : \alpha \to \mathbb{B}.\ \forall\, q : \alpha \to \beta \to \mathbb{B}. \\ &\quad p \stackrel{\star}{\to} q = \lambda\, f : \alpha \to \beta.\ \forall\, x : \alpha.\ x \in p \Rightarrow f\ x \in q\ x \end{aligned} \tag{16}$$

The difference here is that q is a parameterized set, where the parameter comes from the set p. This allows us to model dependent predicate subtypes with predicate sets, such as the following subtype containing natural number subtraction:

$$\mathcal{U}_{\mathbb{N}} \stackrel{\star}{\to} (\lambda\, n.\ \mathsf{lenum}\ n \dot{\to} \mathsf{lenum}\ n)$$

Recall that $\mathcal{U}_{\mathbb{N}}$ is the universe set for the type $\mathbb{N}$ of natural numbers. We should therefore read the above subtype as the set of functions that: given any natural number n return a function from $\{0, \ldots, n\}$ to $\{0, \ldots, n\}$. One last thing to note: if the parameterized set q is of the form $\lambda\, x.\ q'$ where q' does not contain any occurrences of the bound variable x, then $p \stackrel{\star}{\to} q = p \dot{\to} q'$: this shows that Definition 16 is more general than Definition 10.

PVS also has two function space subtypes, covering the dependent and non-dependent cases. It also allows dependent products, which cannot be expressed using our pair notation. However, analogously to $\stackrel{\star}{\to}$ it would be simple to define a dependent pair constructor $\mathsf{dpair}\ p\ q$, taking a set $p : \alpha \to \mathbb{B}$ and a parameterized set $q : \alpha \to \beta \to \mathbb{B}$.

2.3 Subtype Rules

Now that we have defined the form of subtype sets, we shall show how to derive subtypes of a HOL term. Given a term t, we say that p is a subtype of t if we can prove the subtype theorem $\vdash t \in p$. This is the major difference between our model and PVS: here subtypes are theorems, whilst in PVS subtypes are types.

Milner's type inference algorithm [11] for simply-typed terms is structural: a single bottom-up pass of a well-formed term suffices to establish the most general type.[7] We also use a single bottom-up pass to derive sets of subtype theorems, though the algorithm is complicated by two factors:

[7] Though not completely avoiding all difficulty: Mairson [10] has shown that the most general simple type of a term can be exponentially large in the size of the term.

– two rules to break down terms (covering function applications and λ-abstractions) are no longer sufficient since we also need to keep track of logical context;
– there is no concept of a 'most general set of subtype theorems',[8] so instead we perform proof search up to some fixed depth and return all the subtype theorems that we can prove.

To keep track of logical context, we create subtype rules similar to the congruence rules of a contextual rewriter. Here are some examples:

$$\vdash \forall c : \mathbb{B}.\ \forall a : \alpha.\ \forall b : \alpha.\ \forall p : \alpha \to \mathbb{B}.$$
$$(c \in \mathcal{U}_{\mathbb{B}}) \wedge (c \Rightarrow a \in p) \wedge (\neg c \Rightarrow b \in p) \Rightarrow (\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b) \in p \tag{17}$$
$$\vdash \forall a, b : \mathbb{B}.\ (b \Rightarrow a \in \mathcal{U}_{\mathbb{B}}) \wedge (a \Rightarrow b \in \mathcal{U}_{\mathbb{B}}) \Rightarrow (a \wedge b) \in \mathcal{U}_{\mathbb{B}} \tag{18}$$
$$\vdash \forall f : \alpha \to \beta.\ \forall a : \alpha.\ \forall p : \alpha \to \mathbb{B}.\ \forall q : \alpha \to \beta \to \mathbb{B}.$$
$$(f \in p \xrightarrow{\star} q) \wedge (a \in p) \Rightarrow f\ a \in q\ a \tag{19}$$
$$\vdash \forall f : \alpha \to \beta.\ \forall p : (\alpha \to \beta) \to \mathbb{B}.$$
$$(\forall x : \alpha.\ f\ x \in p\ x) \Rightarrow (\lambda x.\ f\ x) \in (\mathcal{U}_{\alpha} \xrightarrow{\star} p) \tag{20}$$

These rules are rather long, but fortunately can be read easily from left to right. For example the conditional subtype rule (17) reads: "if we can show c to be in $\mathcal{U}_{\mathbb{B}}$; and assuming c we can show a to be in a subtype p; and assuming $\neg c$ we can show b to be in the same subtype p; then the combined term $\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b$ must also be an element of p." In this way we can build up logical context. Note that c is trivially in the universe set $\mathcal{U}_{\mathbb{B}}$, the only purpose of retaining this condition is to force the subtype checker to recurse into c and check all its subterms. The conjunction rule (18) similarly ensures that subterms are covered by the subtype checker, while building the correct logical context.[9] Also shown are the subtype rules for function application (19) and abstraction (20), the main point to note is that they both use the more general dependent version $\xrightarrow{\star}$ of the subtype constructor for function spaces.

For each constant that propagates logical information, we need a subtype rule of the above form. Therefore the set of subtype rules used is not fixed, rather we allow the user to add rules for new constants.

[8] Theoretically we could intersect all subtypes that a term t satisfies, but then we would just end up with $\{t\}$ if the logical context was consistent, or $\emptyset$ if it was not!

[9] A version of the conjunction rule that does not always return the universe set $\mathcal{U}_{\mathbb{B}}$ is as follows:

$$\vdash \forall a, b : \mathbb{B}.\ \forall p, q : \mathbb{B} \to \mathbb{B}.\ (b \Rightarrow a \in p) \wedge (a \Rightarrow b \in q) \Rightarrow (a \wedge b) \in (\{\bot\} \cup (p \dot{\wedge} q))$$

where $\dot{\wedge}$ is a lifted version of $\wedge$ that operates on sets of booleans instead of booleans. However, the version we present is much simpler to work with and usually all that is required in practice.

Subtype rules tell us how to derive subtypes for a term by combining the subtypes of smaller terms, but they leave two questions unanswered: how do we calculate the subtypes of base terms (variables and constants); and how do we unify the (possibly higher-order) subtypes of the smaller terms, for example to match the two occurrences of p in the antecedent of the conditional subtype rule (17)? These questions are answered in the next two sections.

2.4 Subtypes of Constants

To calculate subtypes of a base term $t : \alpha$, we maintain a dictionary of constant subtypes.[10] If the term we are focussed on is a constant that appears in the dictionary, we return the subtype theorem listed there. If the term is a variable or a constant that is not in the dictionary, we return the default subtype theorem $\vdash t \in \mathcal{U}_\alpha$.[11]

Here are some miscellaneous entries in the dictionary:

$$\vdash \mathsf{inv} \in (\mathsf{nzreal} \dot{\to} \mathsf{nzreal} \ \cap \ \mathsf{posreal} \dot{\to} \mathsf{posreal} \ \cap \ \mathsf{negreal} \dot{\to} \mathsf{negreal}) \tag{21}$$

$$\vdash \mathsf{sqrt} \in (\mathsf{nnegreal} \dot{\to} \mathsf{nnegreal} \ \cap \ \mathsf{posreal} \dot{\to} \mathsf{posreal}) \tag{22}$$

$$\vdash \forall n : \mathbb{N}.\ - \in \mathcal{U}_\mathbb{N} \stackrel{\star}{\to} (\lambda n.\ \mathsf{lenum}\ n \dot{\to} \mathsf{lenum}\ n) \tag{23}$$

$$\vdash \forall p : \alpha \to \mathbb{B}.\ \mathsf{funpow} \in (p \dot{\to} p) \dot{\to} \mathcal{U}_\mathbb{N} \dot{\to} p \dot{\to} p \tag{24}$$

$$\vdash \forall p : \alpha \to \mathbb{B}.\ [\,] \in (\mathsf{list}\ p \cap \mathsf{nlist}\ 0) \tag{25}$$

$$\vdash \forall p : \alpha \to \mathbb{B}.\ \forall n : \mathbb{N}.\ \mathsf{cons} \in p \dot{\to} (\mathsf{list}\ p \cap \mathsf{nlist}\ n) \dot{\to} (\mathsf{list}\ p \cap \mathsf{nlist}\ (\mathsf{suc}\ n)) \tag{26}$$

$$\vdash \forall f : \alpha \to \beta.\ \forall p : \alpha \to \mathbb{B}.\ \forall q : \beta \to \mathbb{B}.\ \forall n : \mathbb{N}.\ \mathsf{map} \in (p \dot{\to} q) \dot{\to} (\mathsf{list}\ p \cap \mathsf{nlist}\ n) \dot{\to} (\mathsf{list}\ q \cap \mathsf{nlist}\ n) \tag{27}$$

$$\vdash \forall p : \alpha \to \mathbb{B}.\ \forall q : \beta \to \mathbb{B}.\ \forall n : \mathbb{N}.\ \mathsf{zip} \in (\mathsf{nlist}\ n \cap \mathsf{list}\ p) \dot{\to} (\mathsf{nlist}\ n \cap \mathsf{list}\ q) \dot{\to} (\mathsf{nlist}\ n \cap \mathsf{list}\ (\mathsf{pair}\ p\ q)) \tag{28}$$

The universal quantification allows variables in the types of constants, and exactly like 'forall types' in functional programming, these generate fresh variables at every instance of the constant.

This dictionary corresponds to the constant judgement mechanism of PVS, whereby the type-checker can be told that for the purpose of calculating type correctness conditions, particular constants are also elements of more specific subtypes than their principal subtype.

[10] It is up to the user to add constant subtypes to the dictionary: as yet there is no mechanism to automatically generate these for newly defined constants, though this further work is briefly discussed in Section 6.

[11] Note that when we come to use the subtypes of t later on, other subtypes may also be deduced from the logical context.

2.5 Subtype Judgements

Suppose we have a subtype rule that we are committed to using, and we have recursively derived subtype theorems for the terms in the antecedent of the rule. We must now deduce[12] from these subtype theorems, aiming to find a consistent set of subtype theorems that is matched by the antecedent of the rule.

Example 1. Suppose our term is $f\ a$ (where f has simple type $\mathbb{R} \to \mathbb{R}$); we are using the function application rule (19); and we have recursively shown that $\vdash f \in \mathsf{nzreal} \xrightarrow{\cdot} \mathsf{nzreal}$ and $\vdash a \in \mathsf{posreal}$. However, in order to apply the rule we must find instantiations of the variables p and q such that

$$(f \in p \xrightarrow{\star} q) \ \wedge \ (a \in p)$$

is a theorem. We present this goal to our prover, which performs bounded proof search and returns some instantiations, one of which corresponds to the following theorem:

$$\vdash (f \in \mathsf{nzreal} \xrightarrow{\star} (\lambda\, x.\ \mathsf{nzreal})) \ \wedge \ (a \in \mathsf{nzreal})$$

Now we can apply the rule to conclude that $\vdash f\ a \in (\lambda\, x.\ \mathsf{nzreal})\ a$, which can in turn be simplified to $\vdash f\ a \in \mathsf{nzreal}$.

In this example, the prover needed to show $a \in \mathsf{posreal} \Rightarrow a \in \mathsf{nzreal}$. Steps like these are achieved using subtype judgements: theorems that are manually added to the top-level logical context, and are available for use in deriving subtypes.[13] These will be theory specific, and can be extended by the user at any time. Examples are:

$$\vdash \mathsf{posreal} \subset \mathsf{nzreal} \tag{29}$$

$$\vdash \forall p, q : \alpha \to \mathbb{B}.\ p \subset q \ \Rightarrow \ \mathsf{list}\ p \subset \mathsf{list}\ q \tag{30}$$

From the example we can see that a suitable prover must be: higher-order to deal with parameterized types; able to find multiple instantiations of a goal ('prolog-style'); and able to perform bounded proof search. Any prover that satisfies these criteria will be able to plug in at this point and enable subtype derivation.

[12] Deciding the logical context in which we should perform this deduction is quite delicate. It is sound to use the current logical context, but not complete. A more careful approach is to use the (possibly larger) logical context of a subterm whenever we manipulate the subtypes of that subterm. In this way if we can deduce $\vdash 1 \in \mathsf{nzreal}$ and $\vdash \neg\top \Rightarrow 0 \in \mathsf{nzreal}$ then we will be able to deduce $\vdash (\mathsf{if}\ \top\ \mathsf{then}\ 1\ \mathsf{else}\ 0) \in \mathsf{nzreal}$ using the conditional rule.

[13] The name 'subtype judgements' was borrowed from PVS, which contains results used for exactly the same purpose.

However, since there are not many provers available that can satisfy all these requirements, we have implemented one to test our subtype derivation algorithm. Robinson [15] presents an approach to higher-order proving by converting all terms to combinatory form.[14] Together with translation to CNF this conversion leaves terms in a normal form that simplifies the writing of automatic proof search tools. For our application we implement a version of model elimination (mostly following Harrison's presentation [5], with some higher-order extensions), since that is able to return multiple instantiations of goals and we can use a simple depth-bound to limit the search. More powerful normalization means that it can compete with the HOL first-order prover `MESON_TAC` on some first-order problems, and results on higher-order problems are basic but promising. It is under active development, and a paper will soon be available describing its operation in more detail.

2.6 Subtype Derivation Algorithm

To summarize this section, we present the complete algorithm to derive subtypes of a term.

Inputs: A term t having simple type α; a logical context C initialized with a set of assumptions and the current subtype judgements; a set R of subtype rules; and a dictionary D of constant subtypes.

Outputs: A set P of subtype theorems.

1. If t is a variable, return $[\vdash t \in \mathcal{U}_\alpha]$.
2. If t is a constant, look in the dictionary D to see if there is an entry (t, p). If so, return $[\vdash t \in p]$, otherwise return $[\vdash t \in \mathcal{U}_\alpha]$.
3. Otherwise find a subtype rule in R matching t.[15] The rule will have the form

$$\vdash \forall \boldsymbol{v}.\ \left(\bigwedge_{1 \le i \le n} \forall \boldsymbol{v_i}.\ a_i[\boldsymbol{v_i}] \Rightarrow t_i[\boldsymbol{v_i}] \in p_i[\boldsymbol{v_i}, \boldsymbol{v}] \right) \Rightarrow t \in p[\boldsymbol{v}] \qquad (31)$$

4. For each $1 \le i \le n$, create the logical context C_i by adding the assumption $a_i[\boldsymbol{v_i}]$ to C and recursively apply the algorithm using C_i to t_i to find a set of subtypes

$$P_i = \{\vdash t_i[\boldsymbol{v_i}] \in p_{i0}[\boldsymbol{v_i}],\ \ldots,\ \vdash t_i[\boldsymbol{v_i}] \in p_{in_i}[\boldsymbol{v_i}]\}$$

5. Find consistent sets of subtypes by calling the following search function with counter $i \leftarrow 1$ and instantiation $\sigma \leftarrow id$.

[14] Many thanks to John Harrison for drawing my attention to this paper.

[15] If there is more than one rule that matches, return the rule that was most recently added to R: this is almost always the most specific rule too.

(a) If $i > n$ then return σ.
(b) Using the subtype theorems in P_i and the logical context C_i, use the higher-order prover to find theorems of the form $\vdash t_i[\boldsymbol{v_i}] \in \sigma(p_i[\boldsymbol{v_i}, \boldsymbol{v}])$.
(c) For each theorem returned, let σ_i be the returned instantiation. Recursively call the depth-first search function with counter $i \leftarrow i + 1$ and instantiation $\sigma \leftarrow (\sigma_i \circ \sigma)$.

6. Each instantiation σ returned by depth-first search corresponds to a specialization of the subtype rule (31) for which we have proven the antecedent. We may thus deduce the consequent by modus ponens, and we add this to the result set P of subtype theorems.

3 Applications

3.1 Predicate Set Prover

An obvious application for the subtype derivation algorithm is to prove set membership goals. Supposing the current goal is $t \in p$, we can derive a set P of subtype theorems for t, and then invoke the higher-order prover once again with the top-level context and the set P to tackle the goal directly.

Example 2. To illustrate the two steps, consider the goal $3 \in \mathsf{nzreal}$.[16] Subtypes are derived for the term 3 (of type $\mathbb{R}$), and the following list of subtype theorems are returned:

$$[\vdash 3 \in \mathsf{K}\ \mathsf{posreal}\ 3,\ \vdash 3 \in \mathsf{K}\ \mathsf{nnegreal}\ 3]$$

(where $\mathsf{K} = \lambda\, x.\ \lambda\, y.\ x$). Next these two theorems are passed to the higher-order prover along with the top-level logical context containing type-judgements, and it quickly proves the goal $\vdash 3 \in \mathsf{nzreal}$.

We package up the predicate set prover into a HOL tactic, which calls the prover with the current subgoal as argument: if successful the resulting theorem will match the subgoal and can be dispatched. We can prove some interesting goals with this tactic:

$$\vdash \mathsf{map}\ \mathsf{inv}\ (\mathsf{cons}\ (-1)\ (\mathsf{map}\ \mathsf{sqrt}\ [3, 1])) \in \mathsf{list}\ \mathsf{nzreal}$$
$$\vdash (\lambda\, x \in \mathsf{negreal}.\ \mathsf{funpow}\ \mathsf{inv}\ n\ x) \in \mathsf{negreal} \dot{\rightarrow} \mathsf{negreal}$$

One optimization that is effective even with this basic tactic is to maintain a cache of the subtypes that have already been derived for HOL constants.[17] For

[16] This is not quite as trivial as it looks, since the real number '3' in HOL is really the complex term real_of_num (numeral (bit1 (bit1 0))).
[17] Here 'constant' means any term having no free variables.

example, the innocuous-looking term '3' used in the above example is actually composed of 4 nested function applications! Rederiving subtypes for constants is unnecessary and inefficient.

Another optimization arises naturally from certain subtype rules, such as the conjunction rule (18), repeated here:

$$\forall\, a, b : \mathbb{B}.\ (b \Rightarrow a \in \mathcal{U}_{\mathbb{B}}) \wedge (a \Rightarrow b \in \mathcal{U}_{\mathbb{B}}) \Rightarrow (a \wedge b) \in \mathcal{U}_{\mathbb{B}}$$

The set $\mathcal{U}_{\mathbb{B}}$ is the universe set $\mathcal{U}_{\mathbb{B}}$ of booleans, so we can immediately prove $\vdash b \Rightarrow a \in \mathcal{U}_{\mathbb{B}}$ and $\vdash a \Rightarrow b \in \mathcal{U}_{\mathbb{B}}$ without recursing into the structure of the subterms a and b. Note that if we were deriving subtypes in order to check the term for subtype correctness then we would be obliged to carry out this recursion step to check a and b, but if our goal is proving set membership then we can safely skip this.

3.2 Proving Conditions During Rewriting

We can use the predicate set prover as a condition prover in a contextual rewriter, and there are several reasons why it is useful to integrate these tools:

- There is a trend to incorporate tools into contextual rewriters because of the automatic subterm traversal and context accumulation. The logical context built up by the contextual rewriter is easily transferred to the predicate set prover, and the subterm traversal allows us to attempt a proof of all occurrences of $t \in p$ in the goal term.[18]
- Many rewrites have side conditions that can be expressed very naturally using restricted quantifiers, and these generate goals for the predicate set prover when the rewrite is applied.
- Subtype judgements, rules and constants can be stored with the simplification set of a theory, thus reducing the administration burden of theory-specific rules.

Here are some miscellaneous rewrite rules that make use of subtype conditions:

$$\vdash\ \forall\, x \in \mathsf{nzreal}.\ x/x = 1 \tag{32}$$

$$\vdash\ \forall\, n.\ \forall\, m \in \mathsf{lenum}\ n.\ m + (n - m) = n \tag{33}$$

$$\vdash\ \forall\, n \in \mathsf{nznum}.\ n\ \mathsf{mod}\ n = 0 \tag{34}$$

$$\vdash\ \forall\, s \in \mathsf{finite}.\ \forall\, f \in \mathsf{injection}\ s.\ |\mathsf{image}\ f\ s| = |s| \tag{35}$$

$$\vdash\ \forall\, G \in \mathsf{group}.\ \forall\, g \in \mathsf{set}\ G.\ \mathsf{id}_G *_G g = g \tag{36}$$

$$\vdash\ \forall\, G \in \mathsf{group}.\ \forall\, g, h \in \mathsf{set}\ G.\ (g *_G h = h) = (g = \mathsf{id}_G) \tag{37}$$

[18] When subtype derivation is applied to a subterm it accumulates context in much the same way as a contextual rewriter. However, despite this similarity, the two tools are orthogonal and are best implemented separately: we tried both approaches.

Using rule 32, a term like $5/5 = 3/3$ is straightforwardly rewritten to $\top$.

An effective optimization for this tool is to make adding assumptions into the subtype logical context a lazy operation. This delays their conversion to combinatory form and CNF normalization until the predicate set prover is invoked on a goal, which might not happen at all.

The last two examples above come from a body of computational number theory that we have recently formalized [6], which provided a test of the utility of our predicate set prover as a condition prover in a contextual rewriter. The properties that were targeted in the development were group membership (e.g., $g *_G h \in \mathsf{set}\ G$), simple natural number inequalities (e.g., $0 < n$ or $1 < mn$) and nonemptiness properties of lists and sets (e.g., $s \neq \emptyset$).

In theory, the architecture laid out in the previous section can establish much more exotic properties than these, but the predicate subtype prover was found to be most useful and robust on these relatively simple properties that come up again and again during conditional rewriting. These properties naturally propagate upwards through a term, being preserved by most of the basic operations, and in such situations the predicate set prover can be relied upon to show the desired condition (albeit sometimes rather slowly). This tool lent itself to more efficient development of the required theories, particularly the group theory where almost every theorem has one or more group membership side-conditions.

If the predicate set prover had not been available, it would have been possible to use a first-order prover to show most of the side-conditions, but there are three reasons why this is a less attractive proposition: firstly it would have required effort to find the right 'property propagation' theorems needed for the each goal; secondly the explicit invocations would have led to more complicated tactics; and thirdly some of the goals that can be proved using our specialized tool would simply have been out of range of a more general first-order prover.

3.3 Debugging Specifications

How can we use our algorithm for deriving subtypes to find errors in a specification? We do this by invoking the algorithm on the specification, and generating an exception whenever the algorithm would return an empty set of subtypes for a subterm.

Consider the following family of specifications:

$$(\mathsf{inv}\ x) * x = 1 \tag{38}$$

$$x \in \mathsf{nzreal} \Rightarrow (\mathsf{inv}\ x) * x = 1 \tag{39}$$

$$\mathsf{inv}\ x \in \mathsf{nzreal} \Rightarrow (\mathsf{inv}\ x) * x = 1 \tag{40}$$

$$\mathsf{inv} \in \mathcal{U}_{\mathbb{R}} \dot{\rightarrow} \mathsf{nzreal} \Rightarrow (\mathsf{inv}\ x) * x = 1 \tag{41}$$

$$\mathsf{inv} \in \mathcal{U}_{\mathbb{R}} \dot{\rightarrow} \mathcal{U}_{\mathbb{R}} \Rightarrow (\mathsf{inv}\ x) * x = 1 \tag{42}$$

The subtype checker will generate an exception for specification 38, complaining that it could not derive a type for the subterm $\mathsf{inv}\ x$. Why does it say this? The exception is raised because the algorithm could not find a consistent set of subtypes for the subterms inv and x, when using the subtype rule (19) for function application. And this we see to be true, because without any additional knowledge of x we cannot show it to be in any of the sets nzreal, posreal or negreal that the subtype (21) of the constant inv demands.

Specification 39 shows the right solution: add a guard to stop x from taking the illegal value of 0. And this solves the problem, the subtype checker can now derive a subtype of nzreal for the subterm $\mathsf{inv}\ x$ (and a subtype of $\mathcal{U}_{\mathbb{B}}$ for the whole term).

This is how we would expect the subtype checker to be used in practice. A specification is entered and subtype checked, the errors are corrected by adding the necessary guards, and only then is verification started. This could potentially save much wasted effort and act as a valuable teaching tool.

Specifications 40–42 represent various attempts to subvert the subtype checker. Specification 40 is a silly attempt: now the $\mathsf{inv}\ x$ in the antecedent fails to subtype check! However, even if the antecedent were added unchecked to the logical context, the consequent would still not subtype check: an extra subtype for $\mathsf{inv}\ x$ does not help at all in the search to find consistent subtypes for inv and x using the rule for function application. Specification 41 steps up a level in the arms race by assuming a subtype for inv, and now this term does subtype check since the higher-order prover just needs to show that $x \in \mathcal{U}_{\mathbb{R}}$: a triviality. However, we may take consolation in the fact that this antecedent is unprovable. Finally Specification 42 is the most worrying attack: the term subtype checks, and we can use Theorem 13 to prove the condition.

4 Logical Limits

The final example of the previous section showed how to subvert the subtype checker that we implement. Unfortunately, this is not just an inadequacy of the subtype checker, but rather an inescapable consequence of reasoning with predicate sets in the HOL logic. Since HOL is a logic of total functions, given any function $f : \alpha \rightarrow \beta$ we can prove the theorem

$$\vdash f \in \mathcal{U}_{\alpha} \dot{\rightarrow} \mathcal{U}_{\beta} \tag{43}$$

since this just expands to $\vdash \forall x.\ x \in \mathcal{U}_{\alpha} \Rightarrow f\ x \in \mathcal{U}_{\beta}$, which is true by the definition of the universal set $\mathcal{U}_{\beta}$.

This means that enforceable predicate subtyping using predicate sets cannot exist as a layer on top of the existing HOL kernel, since Theorem 43 is true even for restricted constants (such as inv), and can be used by the subtype checker to allow the application of such constants to any argument.

Example 3. Even if the user is not trying to subvert the system, it might happen accidentally. If we are subtype checking the following specification

$$P \Rightarrow Q\ (\mathsf{inv}\ 0)$$

then when we subtype check the consequent $Q\ (\mathsf{inv}\ 0)$ we add the antecedent P to the logical context, and it might transpire that P somehow causes the higher-order prover to deduce $\mathsf{inv} \in \mathcal{U}_{\mathbb{R}} \dot{\to} \mathcal{U}_{\mathbb{R}}$,[19] which then allows $Q\ (\mathsf{inv}\ 0)$ to be successfully subtype checked.

PVS is also a logic of total functions, but the ability to make a first-class type of non-zero reals means that if inv is declared to have type $\mathbb{R}^{\neq 0} \to \mathbb{R}$ then the type-system can stop the function from being 'lifted' to a larger type. 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 all questions of definedness.

5 Conclusion

We have shown how predicate subtyping can be modelled in HOL using predicate sets, explained how to perform subtype checking using our framework, and illustrated some applications of this to specification and verification in HOL.

It is helpful to divide the benefits of predicate subtyping into two categories: negative features, which consist of spotting and disallowing type-incorrect terms; and positive features, which consist of allowing properties to be (more quickly) deduced that help prove theorems.

In this paper we have shown that we can gain many of the positive benefits of predicate subtyping,[20] and in some ways we can even do better: our system does not only calculate with principal subtypes but rather with any subtype that the term can be shown to satisfy using the rules. This was shown to provide an effective proof procedure on a body of formalized mathematics.

[19] After all, it is a theorem!

[20] Although almost certainly the performance using our model is worse than genuine predicate subtyping: keeping subtypes with the terms so they never have to be derived must provide a boost. We also experimented with this kind of architecture, but dropped it in favour of the system presented in the current paper to simplify the design and increase interoperability with existing tools.

In our experience the model is also quite robust when used for the negative features of predicate subtyping, and finds most typical errors that predicate subtyping is designed to prevent. Unfortunately, we have shown that in certain situations it can be subverted, and so we cannot claim to match the PVS level of type safety. However, it is worth remarking that specifications that are provable must possess this level of type safety,[21] so a guarantee is most valuable if the specification will never be verified. If verification is our goal, then any debugging that can be performed in advance will speed up the process, but an absolute guarantee is not necessary. In conclusion, predicate subtyping using predicate sets should be seen as an extra box of tools to aid verification.

6 Further Work

On the basis of our case study, we can seek to improve the predicate set prover by making it faster and more robust on simple and ubiquitous subtypes. One obvious approach to this would be to improve the underlying higher-order prover. In particular the handling of equality and reasoning with total orders could be much more effective; perhaps we could interface to a linear decision procedure to speed up the latter.

Another, more speculative, line of research would be to use subtype checking to perform subtype inference of new constants. If it could be made to work this would be extremely useful: currently for each constant which we would like to add to the constant subtype dictionary, we must specify and prove a result of the form of Theorems 21–28. The idea is to initially enter the type of the new constant c as $c \in p$ where p is a variable; during subtype checking we collect constraints on p; and finally at the top-level we try to solve these constraints: the solutions being subtypes for c.

7 Related Work

The model of predicate subtyping using predicate sets builds upon Wong's [18] restricted quantifier library in HOL88, and the exact details of the predicate subtyping in our model attempts to follow the PVS architecture. For the full details of the semantics and implementation of subtyping in PVS, refer to Owre [14].

Previous work in this area has been done by Jones [7], who built a tool in HOL to specify the subtype of constants and subtype check terms with respect to the subtypes. Given a term t, the tool sets up HOL goals that, if proved, would correspond to the term being subtype-correct. The user is then free to use these

[21] That is, if restricted constants are underspecified outside their domain.

extra theorems during verification. Our model extends Jones' by the introduction of subtype rules for generating type-correctness conditions, the higher-order prover to automatically prove conditions, and the integration of the tool into a rewriter to aid interactive proof.

A comparison of HOL and PVS was made by Gordon [3], from the perspectives of logic, automatic proof and usability. Our work is only relevant to part of this comparison, and enthusiastically takes up some of the suggestions for modelling PVS-style constructs in HOL.

ACL2 uses guards [8] to stop functions from being applied outside their domain; these generate proof obligations when new functions are defined in terms of guarded functions. When the proof obligations have been satisfied the new function is given a 'gold' status, and can be safely executed without causing run-time type errors. This is very similar to the way PVS type-checks terms before admitting them into the system.

Saaltink [16] has also implemented a system of guard formulas in Z/EVES, which both aids formal Z proofs and has found some errors in Z specifications. The Z logic allows terms to be 'undefined', but the system of guard formulas imposed will flag the situations that can result in undefinedness, allowing classical reasoning on the partial logic. Since Z is based on set theory, this use of guards does not suffer from the logical limitations we outlined in Section 4, and can provide strong guarantees about a checked specification. However, whereas our subtype rules propagate all available logical information around the term, Saaltink chooses a "left-to-right system of interpretation" that is not complete, but works well in most practical situations and simplifies guard conditions.

Finally, there has been a huge amount of work on subtyping and polymorphism in various λ-calculi, used to model object-orientated programming languages. Some concepts from this field are related to our work, in particular the notion of intersection types corresponds to finding multiple subtypes of a term. Campognoni's thesis [2] provides a good introduction to this area.

Acknowledgements

My Ph.D. supervisor, Mike Gordon, got me started on this topic and helped bring the project to a successful conclusion. I had many valuable conversations about predicate subtyping with Judita Preiss, Konrad Slind and Michael Norrish, and their comments on this paper (particularly Konrad's keen eye) helped remove many potential misunderstandings. John Harrison gave me help with implementing the higher-order prover, and Ken Larsen should be credited as the local ML guru. Finally, the comments of the TPHOLs referees improved this paper tremendously.

References

1. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
2. Adriana B. Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, Catholic University, Nigmegen, January 1995.
3. M. J. C. Gordon. Notes on PVS from a HOL perspective. Available from the author's web page, 1996.
4. M. J. C. Gordon and T. F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.
5. John Harrison. Optimizing proof search in model elimination. In Michael A. McRobbie and John K. Slaney, editors, *13th International Conference on Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 313–327, New Brunswick, NJ, USA, July 1996. Springer.
6. Joe Hurd. Verification of the Miller-Rabin probabilistic primality test. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, number EDI-INF-RR-0046 in University of Edinburgh Informatics Report Series, pages 223–238, September 2001.
7. Michael D. Jones. Restricted types for HOL. In Elsa L. Gunter, editor, *Supplemental Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics, TPHOLs '97*, Murray Hill, NJ, USA, August 1997.
8. Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.
9. Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.
10. Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 382–401. ACM SIGACT and SIGPLAN, ACM Press, 1990.
11. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.
12. Abdel Mokkedem and Tim Leonard. Formal verification of the Alpha 21364 network protocol. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference: TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 443–461, Portland, OR, USA, August 2000. Springer.
13. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
14. Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.
15. J. A. Robinson. A note on mechanizing higher order logic. *Machine Intelligence*, 5:121–135, 1970.
16. Mark Saaltink. Domain checking Z specifications. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM' 97: Fourth NASA Langley Formal Methods Workshop*, pages 185–192, Hampton, VA, September 1997.

17. Natarajan Shankar and Sam Owre. Principles and pragmatics of subtyping in PVS. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT '99*, volume 1827 of *Lecture Notes in Computer Science*, pages 37–52, Toulouse, France, September 1999. Springer.
18. Wai Wong. *The HOL* `res_quan` *library*, 1993. HOL88 documentation.