Predicate subtyping in VerCors

Tycho Dubbeling
t.b.dubbeling@student.utwente.nl
University of Twente
Enschede, The Netherlands

ABSTRACT

VerCors is a program verifier that can verify specifications for programs written in Java, C and PVL (VerCors own language with built-in support for VerCors specifications). These specifications are expressed through pre-conditions and post-conditions similar to JML (Java Modeling Language) annotations.
Many Java and C programs use bounded integers, rather than mathematical integers. Meanwhile, VerCors can only reason about all integer-based types as mathematical integers. As such, VerCors cannot verify a program for the absence of integer overflows.
Bounded integers can be seen as subtypes of integers. Predicate subtyping is a subtyping system in which a subtype gets constructed with a type and predicates on a term of said type. The terms of the constructed subtype are all terms of the original type for which all predicates hold. We have added predicate-based subtyping to VerCors as a more general solution to supporting bounded integers. In this thesis we will first show how to manually translate examples of the proposed subtyping system into the corresponding specification in a program. We will also establish supporting mathematical theory describing the relations between predicate subtypes.

KEYWORDS

VerCors, predicate subtyping, bounded integers, validation

1 INTRODUCTION

Java and C programs commonly use bounded integers. The presence of integer overflows may introduce undefined behaviour or crashes into a program. Undefined behaviour may alter the visible behaviour of programs in unforseen ways, potentially causing unexpected behaviour by the program. In this thesis we will be using the term integer overflow to refer to instances of arithmetic with bounded integers resulting in values outside of said bounds.
Here is an example of an integer overflow from the C programming language: an unsigned integer x (integer with a non-negative value) will overflow when "x = 0; x = x-1;" gets executed. Integer overflow may also occur when typecasting an integer to an integer type with stricter bounds.
Since integer overflows may cause a program to not behave as intended, we would like to have a way to show the absence of potential overflows in our program. One way this may be achieved is through the use of external tooling.
The VerCors verifier, developed at the University of Twente, is software for verifying partial correctness of annotated Java and C programs. VerCors also supports verifying its own language PVL, which has VerCors annotations built into the language. Since absence of integer overflows are a property that may be shown for any arithmetic expression in a program, VerCors can be a useful tool for proving the absence of overflows. Currently, VerCors treats all integer types as mathematical integers and lacks support for bounded integers. As such, VerCors does not support verifying the absence of integer overflows out of the box. This is something we wish to rectify, as having bounded integer support in VerCors will allow VerCors to prove the absence of another class of potentially undesired behaviour.
One way of viewing bounded integers is as a subtype of integers with all its values between two mathematical integers. As a result, the restrictions for bounded integers lend themselves well to being modelled using predicates [13]. However, manually specifying the bounds for every instance of arithmetic being performed is time-consuming and error-prone. With the addition of predicate subtyping, this may be automated. The addition of predicate subtyping in VerCors allows bounded integers to be modelled with a more general system, alongside allowing extension of VerCors with more user-defined subtypes. The user-made subtypes allow users to encode invariants specific to their programs as subtypes, allowing a way to automatically generate the annotations to enforce this invariant throughout the code. For example, a user may want to make sure that a integer variable that is used for division is never zero. The user could then declare a subtype "NonZero" that would assert that the variable does not equal 0 and then annotate the relevant variables with the subtypes.
We have the following research question:

How can a predicate subtyping system capable of encoding bounded integers be added to VerCors?

To answer the main question, we will need to answer more specific questions as well. One question we will try to answer is how to translate subtype annotations into first-order logic VerCors annotations. We will also discuss how predicate subtyping may be implemented into VerCors. Finally, we will work out a mathematical background for the "is a subtype of" relation between predicate subtypes. Our main contributions are showing how to encode predicate subtyping in VerCors, using an approach that may be applied to other verifiers as well, and working out a category-theoretical description of the subtyping relations induced by predicate subtyping.

TScIT 41, July 05, 2024, Enschede, The Netherlands
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND

VerCors uses annotations similar to Java Modeling Language as specification for a program. JML [11] is a language for writing specifications for Java programs, using assertions on predicates written in first-order logic. These specifications get used by many tool to prove properties about a program using Hoare logic [12]. Hoare logic primarily uses pre-conditions and post-conditions to reason about the change of program-state after a command. Unlike JML, VerCors uses separation logic for verification. Separation logic is an extension of Hoare logic that keeps track of state throughout the program and introduces operators that make assertions about the state of the program. The ability to reason about read-write permissions in particular makes separation logic suitable for verifying concurrent programs.
VerCors tries to verify assertions about programs which are made in the specification of said program. A programmer may specify conditions that must hold at a specified position within the program. For example, these assertions may be pre-conditions and post-conditions for functions or assertions about the state of the program at a specific step of execution. VerCors will try to prove that all pre-conditions hold when a function gets called in a function to be proven. Similarly, it wil try to prove that the post-conditions of the function hold as well. An example of such a function with specification:

```
public class Example {
    //@ requires y != 0;
    //@ ensures \result == x/y;
    private int  division(int x, int y) {
        return x/y;
    }
}
```

The process VerCors goes through during verification can be described through 5 major phases. First VerCors parses the program. The grammar VerCors parses is built out of a specification grammar and a grammar for the language it needs to parse. After VerCors finishes parsing, it will proceed to build a syntax tree out of the parsed nodes. During this second step, VerCors may already produce an error if it detects an unsupported feature in the parsed nodes. As a third step, VerCors will try to resolve references and types within the parsed program. After the resolution step, VerCors will proceed to the transformation step. During the transformation step, VerCors will apply a list of rewriters to reduce the syntax tree into a form that is accepted by the back-end. The final step is the step where the rewritten syntax tree gets given to the Viper back-end for verification. In figure 1 these phases are visualized. The creation of the syntax tree, resolution step and the transformation step are all bundled together in the "transformation" arrow.

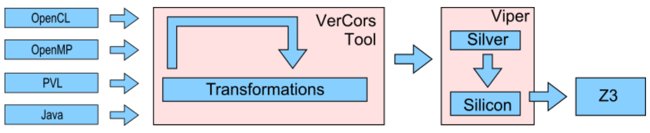

Figure 1: Verification flowchart

3 RELATED WORK

There has been previous work on encoding bounded integers in program verifiers built on Viper. One such verifier is Prusti [1] for the Rust language, which attempts to prove the absence of integer overflows.
Below are an explanation of Viper syntax and examples of how Prusti encodes 32-bits signed integers in Viper [13].
Viper has a built-in datatype called Ref. The value of a Ref is either a pointer to an object or the constant null.
A field declares a variable of a type that is a component of every object. Fields are declared at top level.
A predicate gives a name to a set of conditions to be proved. A predicate declaration may take arguments that may be used in the conditions. Predicates may be attached to methods as pre- and post-requisites. Viper will try to prove all pre- and post-requisites of a method.
acc is a predicate asking for write permission for a variable.
Prusti encodes integers as:

```
field val_int
predicate i32(self: Ref) {
  acc(self.val_int, write)
}
```

and bounded integers with overflow checks as:

```
predicate i32(self: Ref) {
  acc(self.val_int, write) &&
    -2147483648 <= self.val_int &&
    self.val_int <= 2147483647
}
```

As can be seen in the examples, Prusti encodes integers as a predicate [16] that takes a Ref type with an integer field. When overflow checks are turned on, Prusti adds the necessary bounds to the predicate. This gives us a nice template for the predicates we could use to simulate bounded integers through predicate subtypes. The existence of prior art in regards to modeling bounded integers with predicates shows that this approach is feasible. Since these predicates can be directly written in VerCors and give us the desired behaviour for verification, we made sure that our design could accommodate this representation.
Another tool for verifying the absence of integer overflows within software is Frama-C. Frama-C is a static analyzer tool for C programs. Frama-C allows the use of plugins, including plugins for checking integer overflows.
The RTE plugin for Frama-C is a plugin that provides verification for pre- and post-conditions for functions with ACSL specifications. The RTE plugin also generates assertions checking for overflows at every downcast [6]. Furthermore, it generates annotations for all arithmetic calculations that were peformed. As such, the RTE plugin automatically verifies whether overflowing operations occur in the analyzed program.
The WP plugin for Frama-C implements a weakest-precondition calculus for verifying annotated C-programs. The WP plugin has different settings for how to treat arithmetic by default [3]. These settings are called models and can be further modified by adding

extra flags. The WP plugin has two "models" for integer arithmetic and two "models" for floating-point arithmetic. For integers, these models govern whether operations on certain datatypes are allowed to overflow. There are two models for integer arithmetic: the machine integer model and the natural model.
The machine integer model assumes that signed integers are not allowed to overflow, while unsigned integer arithmetic is allowed to overflow. For verification purposes, arithmetic that is allowed to overflow gets interpreted as modular addition. Meanwhile, types that are not allowed to overflow will have all arithmetic performed on them be interpreted as their respective mathematical operations. Casts between integer types use the modulo operation to determine the result of the cast.
The natural model uses mathematical operations on all integer types. Boundedness conditions are dropped, except for unsigned integers being non-negative. Dropping the bounds prevents expensive calculations by the prover. Casts between integer types are still interpreted with modular arithmetic.
The RTE and WP plugins are examples of prior work on where assertions should be generated to detect integer overflows. Furthermore, they highlight the possibility of adding a setting to VerCors to automatically add the bound-checking predicates. The natural model of the WP plugin reasons about integers the same way as VerCors currently does. The considerations for both models of reasoning highlight one of the trade-offs for full verification for the absence of integer overflows: verification time. Since verification time may inhibit programmer productivity by way of delayed results, we strive to make both modes of reasoning possible. To accommodate the possibility of running VerCors with both the possibility of treating bounded integers as mathematical integers and as proper bounded integers, our predicate subtyping design has to take into account the possibility of backwards compatible behaviour with the current mode of reasoning for VerCors.
There has been previous work on implementing predicate subtyping. For example, the PVS verification system has implemented predicate subtyping using the verification abilities already provided by PVS [14] [7] [8].
Much mathematical theory has been developed as well for subtyping. Notably, categorical models that embed a notion of subtyping have been developed [4] [2]. However, these describe more general notions of subtypes. Meanwhile, we will be focusing specifically predicate subtypes.

4 EXAMPLES AND TRANSLATIONS

In this section we first introduce the guidelines we used for determining the rewrite rules for subtype annotations, after which we describe the rewrite rules in a language-agnostic way. Finally, this section will finish with two examples of programs with subtype annotations and a corresponding program with those annotations rewritten into their corresponding specifications.
There are multiple abstract rewrite rules for rewriting subtype annotations into verification annotations. These rewrite rules are motivated by the goal of making sure a subtyped variable/expression always fulfill its subtyping predicates at any point that it may be accessed. This goal induces the following rewrite rules:

- subtypes on function/method input parameters should add pre-conditions to the function/method asserting that the subtyped parameters fulfill the relevant predicates.
- subtypes on the return type of a function/method should add post-conditions to the function/method asserting that the return value of the function/method satisfies the subtype predicates.
- expressions/statements that assign a value to a subtyped variable/a field of a subtyped variable should have assertions added after the assignment that assert that the new value of the variable satisfies the subtype predicates.

Note that this list is not necessarily exhaustive, as different programming languages may have different features for mutating values. We have chosen for generating assertions after variable assignments because of ease of implementation in VerCors. An alternative method would be adding an assertion before the assignment that checks the result of the assignment expression. Both options would prevent the variable from reaching an invalid state, meaning the list of rewrite rules may be amended to include assertion generation before assignments instead. Another choice we made was to inline the subtype predicates, rather than requiring them to be unfolded like predicates in VerCors. The reason we made this choice is because the subtyping system may generate a lot of proof obligations for subtyped variables. Since there may be quite a few assertions about the subtype, we wanted to avoid requiring unfolding the subtype for basic operations that require simple proofs. One example of such an operation would be indexing an array. VerCors requires a value used for indexing to be smaller than the array length. If a variable is subtyped to be within that range, it would need to be unfolded every time an array gets indexed. For this reason, we decided to make the default behaviour inline the subtype predicates.
Another comment on the rewrite rules: type casts for integer types should be treated as functions with its input argument subtyped with a predicate corresponding to the bounds of the type that is casted to.
To add multiple subtype restrictions to a type, the relevant subtype names may be written with spaces between the name. These will be converted into a conjugated predicate, since the variable must adhere to all the subtype predicates.
The second list of parameters for the subtype declarations are to be substituted with constant values. These constant values must be given when adding the subtyping restriction to a type. In the below code example the "Index" subtype is an example of a subtype with extra parameters. These parameters allow abstracting over constant values, which may be used in cases where the difference between subtype predicates may be a constant value.
Below is a Java example using VerCors annotations. We numbered the lines with subtype applications in the example and gave the corresponding line numbers for the corresponding specifications in the rewritten version of the examples:

```
public class SubtypingExample {
    //@ subtype NonZero(int x)() = x != 0;
    /*@ subtype Byte(int x)() = x >= -128
            && x <= 127; @*/
    //@ subtype NonNull(int[] x)() = x != null;
    /*@ subtype Index(int x)(int length) =
    x < length; @*/

    //@ ensures \result == x/y;
    public static int division(int x,
1     /*@ NonZero @*/ int y) {
        return x/y;
    }

    public static void main(String[] args) {
2       /*@ Byte @*/ int result = division(3,5);
3       /*@ NonNull @*/ int[] array = new int[4];
4       /*@ Index(array.length) @*/ int index = 3;
        array[index] = result;
    }
}
```

This example would be rewritten into:

```
public class SubtypingExample {

1   //@ requires y != 0;
    //@ ensures \result == x/y;
    public static int division(int x, int y) {
        return x/y;
    }

    public static void main(String[] args) {
        int result = division(3,5);
2       //@ assert result >= -128 && result <= 127;
        int[] array = new int[4];
3       //@ assert array != null;
        int index = 3;
4       //@ assert index < array.length.
        array[index] = result;
3       //@ assert array[index] != null;
        // asignment to a field causes check
    }
}
```

Another example in Java, but with multiple subtypes annotated at a single variable:

```
public class NonNullVectors {
    //@ subtype nonNull(int[] xs)() = xs!=null;
    //@ subtype len2(int[] xs)() = xs.length==2;
    //@ subtype len3(int[] xs)() = xs.length==3;

    //@ requires Perm(xs[*],1);
    //@ ensures Perm(xs[*],1);
    private static void swap(
1     /*@ nonNull len2 @*/ int[] xs) {
        int left = xs[0];
        int right = xs[1];
        xs[0] = right;
        xs[1] = left;
    }

    //@ requires Perm(xs[*],1) ** Perm(ys[*],1);
    //@ ensures Perm(xs[*],1) ** Perm(ys[*],1);
4   private static /*@ nonNull len3 @*/ int[] cross(
2     /*@ nonNull len3 @*/ int[] xs,
3     /*@ nonNull len3 @*/ int[] ys) {
        int left = xs[1]*ys[2] - xs[2]*ys[1];
        int middle = xs[2]*ys[0] - xs[0]*ys[2];
        int right = xs[0]*ys[1] - xs[1]*ys[0];
        return new int[]{left,middle,right};
    }
}
```

which will be rewritten to:

```
public class NonNullVectors {

1   //@ requires xs!=null
1   //@ requires xs.length==2;
    //@ requires Perm(xs[*],1);
    //@ ensures Perm(xs[*],1);
    private static void swap(int[] xs) {
        int left = xs[0];
        int right = xs[1];
        xs[0] = right;
        xs[1] = left;
    }

2   //@ requires xs!=null
2   //@ requires xs.length==3;
3   //@ requires ys!=null
3   //@ requires ys.length==3;
    //@ requires Perm(xs[*],1) ** Perm(ys[*],1);
4   //@ ensures \result!=null
4   //@ ensures \result.length==3;
    //@ ensures Perm(xs[*],1) ** Perm(ys[*],1);
    private static int[] cross(int[] xs, int[] ys) {
        int left = xs[1]*ys[2] - xs[2]*ys[1];
        int middle = xs[2]*ys[0] - xs[0]*ys[2];
        int right = xs[0]*ys[1] - xs[1]*ys[0];
        return new int[]{left,middle,right};
    }
}
```

Both rewritten versions of the examples above should verify. Our implementation is capable of properly verifying both examples as expected.

5 IMPLEMENTATION

We managed to implement a prototype of the rewrite rules in VerCors. We have added subtyping annotations to the Java parser for VerCors and subtype declarations to the specification parser for VerCors. Furthermore, we have added a rewrite step to the transformation phase of VerCors. Currently, the prototype implementation implements the function contract rewrite rules for methods and generates an assert at variable assignments. The prototype also implements annotating a type with multiple subtypes. Finally, we

managed to implement the constant parameters for subtype applications like in the "Index" subtype in the examples from the previous section.
Because of time constraints we were not able to implement all the planned features. We did not manage to add assertions for variables declared in forall/exists clauses. We also do not yet check for side effects in object methods upholding the subtype predicates. Finally, we did not manage to implement a setting to enable subtyping for bounded integers by default, meaning that VerCors will still treat bounded integers as mathematical integers when no annotations are present.
We do not expect any conceptual problems for implementing these missing features. The conceptual groundwork for their implementation is already present and just needs engineering time spent on implementing them.

6 THEORY

This section is split into three subsections. First we will be constructing a preorder to express a subtyping relation for predicate subtypes. Then we will be converting the preorder into a category, using said category to show how certain subtypes for a fixed type relate to each other. In the third subsection we will finish by showing how certain subtype-categories relate to each other.

The preorder of subtypes of a type

We will start this section by identifying a preorder on the set of all predicate subtypes of a type. This will give us confidence that predicate subtyping does indeed act like a subtyping relation. For this section we will assume the logic used to be classical predicate logic, meaning that we assume the law of excluded middle ($\neg P \lor P$ is always true). As a refresher, a preorder on a set **P** is a binary relation **R** for which the following hold:

Definition 6.1.

- **R** is transitive ($\forall a, b, c \in \mathbf{P}. a \leq b \land b \leq c \Rightarrow a \leq c$)
- **R** is reflexive ($\forall a \in \mathbf{P}. a \leq a$)

We find that we can construct such a relation for all predicate subtypes of any fixed type.
We will first establish a notational convention for predicate subtypes. The tuple $(pred, A)$, where A is the base type and $pred$ the subtype predicate, will be abbreviated as A. When writing $a : A$ we take it to mean that a is an element of the base type of A for which the predicate expression holds true. We will use $\mathcal{P}_A$ to denote the predicate carried by the predicate subtype A. We will only consider subtypes up to equivalence of predicates, meaning that $A \cong B$ if and only if $\mathcal{P}_A$ is equivalent to $\mathcal{P}_B$ in predicate logic.
We will now define the previously mentioned subtyping relation S to be $\{(A, B) | \forall a : A. a : B\}$. This may be read as "$(A, B) \in$ S if every term of A is a term of B". Since every admitted term of A is a term of A by definition, we know that S is reflexive. We also know that the relation S is transitive, since

$$(\forall a : A. a : B) \land (\forall b : B. b : C) \Rightarrow \forall a : A. a : C$$

One example of such a preorder is the preorder of subtypes on the boolean type as seen in Figure 2:

$$\begin{tikzcd}
 & Bool & \\
isFalse \arrow[ur, "includes"] & & isTrue \arrow[ul, "includes"'] \\
 & empty \arrow[ul, "includes"] \arrow[ur, "includes"'] &
\end{tikzcd}$$

Figure 2: Boolean subtype preorder

We only considered a subset of possible predicate subtypes of boolean for brevity sake.

The category of subtypes of a type

Now that we have obtained a preorder on the subtypes of a type, we can construct a corresponding category [5]. We will use this category to show that we can deduce certain subtype relations from other subtype relations. One thing we will show that we can construct a common supertype and a common subtype for all subtypes of a base type. We will also show a generic way of finding a least common supertype (respectively subtype) for two subtypes of a type. Furthermore, we will show that we have a notion of currying for subtyping relations as described in the previous subsection.
A category C is a tuple composed of:

Definition 6.2.

- a class of objects $Ob(C)$
- for each pair of objects $x, y \in Ob(C)$ there is a class $Hom_C(x, y)$ whose elements are called morphisms (from x to y)
- for each object x there is an unique identity morphism id_x in $Hom_C(x, x)$
- a composition map

  $$\circ : Hom_C(x, y) \times Hom_C(y, z) \rightarrow Hom_C(x, z)$$

  where $x, y, z \in Ob(C)$

This composition map must be associative

$$(f \circ g) \circ h = f \circ (g \circ h)$$

and identity morphisms must act like identity elements for the composition map

$$id_x \circ f = f \land f \circ id_y = f$$

We will now translate the preorder **S** on the set of subtypes of a type X, henceforth denoted as $Sub(X)$, to a category $\mathcal{S}(X)$:

- let $Ob(\mathcal{S}(X)) = Sub(X)$
- if $(x, y) \in$ S
  then let $Hom_{\mathcal{S}(X)}(x, y) = \{(x, y)\}$
  else $Hom_{\mathcal{S}(X)}(x, y) = \{\}$
- let $\circ((x, y), (y, z)) := (x, z)$

We find that every object in $\mathcal{S}(X)$ has an identity morphism because S is reflexive and

$$\forall x, y \in Ob(\mathcal{S}(X)).\ (x, x) \circ (x, y) = (x, y) = (x, y) \circ (y, y)$$

Furthermore, we find that $\forall a, b, c, d \in Ob(\mathcal{S}(X))$:

$$((a, b) \circ (b, c)) \circ (c, d) = (a, d) = (a, b) \circ ((b, c) \circ (c, d))$$

We also find that for any two objects $x, y \in Ob(\mathcal{S}(X))$, the existence of morphisms both ways imply that the objects are isomorphic. This can be seen by the equations

$$(x, y) \circ (y, x) = (x, x) = id_x$$

and

$$(y, x) \circ (x, y) = (y, y) = id_y$$

Isomorphism in $\mathcal{S}(X)$ means that both subtypes admit the same elements, giving us a sanity check that our category makes sense for what we which to study through it.

$$\begin{tikzcd}
Bool \arrow[rr, shift left] & & isFalse||isTrue \arrow[ll, shift left] \\
isFalse \arrow[u] \arrow[urr] & & isTrue \arrow[u] \arrow[ull] \\
& empty \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 3: Boolean subtype category

Now that we have constructed a category $\mathcal{S}(X)$, we are interested in the properties it satisfies. Particularly, we will want to show that $\mathcal{S}(X)$ is a cartesian closed category. Showing that $\mathcal{S}(X)$ is a cartesian closed category will tell us that for any morphism $f : A \times B \to C$ we have unique corresponding morphism $g : A \to C^B$. In more familiar categories like the category of sets, this correspondence is often called currying.

Definition 6.3. A category C is cartesian closed if:

- C has a terminal object
- any two objects $A, B \in Ob(C)$ have a product $A \times B$ in $Ob(C)$
- any two objects $A, B \in Ob(C)$ have an exponential object A^B in $Ob(C)$

The first two properties we wish to show are the existence of an initial object and a terminal object. The existence of these objects tells us that there is a subtype that is the supertype of all subtypes of X (terminal object) and that there is a subtype that is a subtype of all subtypes of X (initial object).

Definition 6.4. In category theory, the initial object is characterized by having exactly one morphism from itself to every other object in the category.

Definition 6.5. Similarly, the terminal object is the object in a category that has exactly one incoming morphism for every object in the category.

We find that, for every type X, the category $\mathcal{S}(X)$ contains a initial and terminal object. The initial object in $\mathcal{S}(X)$ is given by the subtype *empty* with the predicate $false$. Since *empty* admits no terms, all of its term are admitted by all subtypes of X. As such, there is a morphism to every other object. As there is either only one morphism (x, y) or no morphisms $\forall x, y \in Ob(\mathcal{S}(X))$, we know that *empty* must have exactly one morphism to all other subtypes of X. Similarly, we find that X with predicate "true" admits every term of X. Since all subtypes of X only admit terms of X, every subtype of X has a morphism to X. For example, in $\mathcal{S}(int)$ the initial object is the empty subtype and the terminal object is int as seen in Figure 4:

$$\begin{tikzcd}
& int & \\
isEven \arrow[ur] & Nat : x \geq 0 \arrow[u] & isOdd \arrow[ul] \\
& empty \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Figure 4: int subtype category

the choice to show the subtypes isEven, isOdd and Nat was arbitrary, since any predicate subtypes of int could have been used to illustrate the concepts of initial and terminal objects.
Another property of interest is the presence of products and coproducts. Taking the product (respectivelly coproduct) of two subtypes will give us a subtype (respectivelly supertype) with a minimally (respectivelly maximally) restrictive predicate for both given subtypes. We will proceed to first give the definition of a product and then show that $\mathcal{S}(X)$ contains the product of any two objects in $\mathcal{S}(X)$.

Definition 6.6. Given a category C, a tuple

$$(p \in Ob(C), \pi_a : p \to a, \pi_b : p \to b)$$

is a product if, given any other tuple

$$(c \in Ob(C), f : c \to a, g : c \to b)$$

there exists a unique morphism

$$(f, g) : c \to p$$

such that both $f = \pi_a \circ (f, g)$ and $g = \pi_b \circ (f, g)$ hold.

$$\begin{tikzcd}
& C \arrow[dl, "f"'] \arrow[d, dashed, "{(f,g)}"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\pi_A"] \arrow[r, "\pi_B"'] & B
\end{tikzcd}$$

Figure 5: Product visualized

We will now show that for any $x, y \in \mathcal{S}(X)$ the product $x \times y$ exists in $\mathcal{S}(X)$. Consider subtypes $Left, Right \in \mathcal{S}(X)$ and the subtype $Middle$ with predicate $\mathcal{P}_{Left} \wedge \mathcal{P}_{Right}$. By predicate logic we know that

$$\mathcal{P}_{Middle} \Rightarrow \mathcal{P}_{Left} \wedge \mathcal{P}_{Middle} \Rightarrow \mathcal{P}_{Right}$$

meaning that $\forall m : Middle$ we find that $m : Left \wedge m : Right$. Given some subtype C, with morphisms $f : C \to Left$ and $g : C \to Right$, we know that

$$\mathcal{P}_C \Rightarrow \mathcal{P}_{Left} \wedge \mathcal{P}_C \Rightarrow \mathcal{P}_{Right}$$

Since $\mathcal{P}_C \Rightarrow \mathcal{P}_{Left} \wedge \mathcal{P}_C \Rightarrow \mathcal{P}_{Right}$ can be rewritten to $\mathcal{P}_C \Rightarrow \mathcal{P}_{Left} \wedge \mathcal{P}_{Right}$, we know that there exists a morphism $(C, Middle)$.

Since for any two objects $x, y \in \mathcal{S}(X)$ there can only be one morphism from x to y, we know that

$$(Middle, (Middle, Left), (Middle, Right))$$

uphold the properties of a product. Since $Left$ and $Right$ were arbitrary objects in $\mathcal{S}(X)$, we find that the product exists for any two objects in $\mathcal{S}(X)$.

$$\begin{tikzcd}
& x >= 0 \ \&\& \ (x \bmod 4) == 0 \arrow[dl] \arrow[d, dashed] \arrow[dr] & \\
x >= 0 & x >= 0 \ \&\& \ (x \bmod 2) == 0 \arrow[l] \arrow[r] & (x \bmod 2) == 0
\end{tikzcd}$$

Figure 6: A product in $\mathcal{S}(int)$

Next up we will give the definition of a coproduct in a category:

Definition 6.7. Given a category C, a tuple

$$(p \in Ob(C), \iota_a : a \to p, \iota_b b \to p)$$

satisfies the universal property of a coproduct if given any other tuple

$$(c \in Ob(C), f : a \to c, g : b \to c)$$

there exists a unique morphism

$$(f|g) : p \to c$$

such that $f = (f|g) \circ \iota_a$ and $g = (f|g) \circ \iota_b$.

$$\begin{tikzcd}
& C & \\
A \arrow[ur, "f"] \arrow[r, "\iota_a"'] & A|B \arrow[u, dashed, "(f|g)"] & B \arrow[l, "\iota_b"] \arrow[ul, "g"']
\end{tikzcd}$$

Figure 7: Coproduct visualized

We find that a coproduct exists for any $x, y \in \mathcal{S}(X)$:
Consider subtypes $Left, Right \in \mathcal{S}(X)$ and the subtype $Middle$ with predicate $\mathcal{P}_{Left} \vee \mathcal{P}_{Right}$.
Since

$$\mathcal{P}_{Middle} = \mathcal{P}_{Left} \vee \mathcal{P}_{Right}$$

we know that

$$\mathcal{P}_{Left} \Rightarrow \mathcal{P}_{Middle} \wedge \mathcal{P}_{Right} \Rightarrow \mathcal{P}_{Middle}$$

For any subtype C with morphisms $f : Left \to C$ and $g : Right \to C$ we know that

$$(\mathcal{P}_{Left} \Rightarrow \mathcal{P}_C \vee \mathcal{P}_{Right} \Rightarrow \mathcal{P}_C) = \mathcal{P}_{Left} \vee \mathcal{P}_{Right} \Rightarrow \mathcal{P}_C$$

This gives us $\mathcal{P}_{Middle} \Rightarrow \mathcal{P}_C$, meaning that $(f|g)$ does indeed exist given any f and g. Like with the case of the product, we find that

$$(f = (f|g) \circ \iota_{Left}) \wedge (g = (f|g) \circ \iota_b)$$

because, for any two objects $x, y \in \mathcal{S}(X)$, the only possible morphism is (x, y). Since $Left$ and $Right$ were arbitrary objects in $\mathcal{S}(X)$, we can construct a coproduct for any two objects in $\mathcal{S}(X)$.

We will now discuss the last property of $\mathcal{S}(X)$ that we will discuss in this subsection, the existence of the exponential object b^a for any $a, b \in \mathcal{S}(X)$. The definition of an exponential object is as follows:

$$\begin{tikzcd}
& x == 1||x == 2||x == 3 & \\
x == 1 \arrow[ur] \arrow[r] & x == 1||x == 2 \arrow[u, dashed] & x == 2 \arrow[l] \arrow[ul]
\end{tikzcd}$$

Figure 8: A coproduct in $\mathcal{S}(int)$

Definition 6.8. Consider the category C with objects $a, b \in Ob(C)$ and let C contain all binary products with b. An exponential object is an object b^a with a morphism

$$f : b^a \times a \to b$$

if, given any object $c \in Ob(C)$ with morphism

$$g : c \times a \to b$$

there is a unique morphism

$$h : c \to b^a$$

such that:

$$f \circ (h \times id_a) = g$$

This equation is visualized in Figure 9:

$$\begin{tikzcd}
C \times A \arrow[d, dashed, "h\times id_A"'] \arrow[dr, "g"] & \\
B^A \times A \arrow[r, "f"'] & B
\end{tikzcd}$$

Figure 9: Exponential visualized

We will now show that we can construct an exponential object for any two objects in $\mathcal{S}(X)$:
Consider objects $Source, Target \in Ob(\mathcal{S}(X))$ and let $Exp \in Ob(\mathcal{S}(X))$ be the subtype with predicate

$$\mathcal{P}_{Source} \Rightarrow \mathcal{P}_{Target} = \neg\mathcal{P}_{Source} \vee \mathcal{P}_{Target}$$

We find that for any such object Exp, we have a morphism

$$f : Exp \times Source \to Target$$

since in classical predicate logic

$$(\neg\mathcal{P}_{Source} \vee \mathcal{P}_{Target}) \wedge \mathcal{P}_{Source} \Rightarrow \mathcal{P}_{Target}$$

We will now continue to show that Exp is an exponential object $Target^{Source}$. Consider some object $Select \in Ob(\mathcal{S}(X))$ and a morphism $g : Select \times Source \to Target$. The existence of g implies that

$$\mathcal{P}_{Select} \wedge \mathcal{P}_{Source} \Rightarrow \mathcal{P}_{Target}$$

We now find that

$$\begin{aligned}
&\mathcal{P}_{Select} \wedge \mathcal{P}_{Source} \Rightarrow \mathcal{P}_{Target} \\
&= \neg(\mathcal{P}_{Select} \wedge \mathcal{P}_{Source}) \vee \mathcal{P}_{Target} \\
&= \neg\mathcal{P}_{Select} \vee \neg\mathcal{P}_{Source} \vee \mathcal{P}_{Target} \\
&= \mathcal{P}_{Select} \Rightarrow \neg\mathcal{P}_{Source} \vee \mathcal{P}_{Target}
\end{aligned}$$

meaning that the existence of g implies the existence of a morphism $h : Select \to Exp$. Because $\mathcal{S}(X)$ has either no morphisms or exactly one morphism from a fixed object to another fixed object, we know that h is unique. As a result, we find that

$$f \circ (h \times id_{Source}) = g$$

As a sanity check, we find that

$$(\mathcal{P}_{Source} \land \mathcal{P}_{Select} \Rightarrow \mathcal{P}_{Exp} \land \mathcal{P}_{Source})$$
$$\land$$
$$(\mathcal{P}_{Exp} \land \mathcal{P}_{Source} \Rightarrow \mathcal{P}_{Target})$$

gives us

$$\mathcal{P}_{Source} \land \mathcal{P}_{Select} \Rightarrow \mathcal{P}_{Target}$$

As such, Exp is an exponential object $Target^{Source}$. Since $Source$ and $Target$ were arbitrary objects in $\mathcal{S}(X)$, we find we can construct an exponential object for any two objects in $\mathcal{S}(X)$.

$$\begin{tikzcd}
x \geq 0 \;\&\&\; x < 8 \arrow[d, dashed] \arrow[dr] & \\
(x < 8 \Rightarrow (x \geq 0 \;\&\&\; x < 8)) \;\&\&\; x < 8 \arrow[r] & x \geq 0 \;\&\&\; x < 8
\end{tikzcd}$$

Figure 10: exponential object in $\mathcal{S}(int)$

Since we have shown that $\mathcal{S}(X)$ has a terminal object, a product for any two objects and an exponential object B^A for any two objects B and A, we have now shown that $\mathcal{S}(X)$ is a Cartesian closed category for any type X [10]. On top of that we have shown an even stronger result: $\mathcal{S}(X)$ is a Bicartesian closed category. A Bicartesian closed category is a Cartesian closed category with an initial object and a coproduct for any two objects in the category [9].

The category of subtype categories

In this subsection we will be studying how $\mathcal{S}(X)$ with different types X relate to each other. We will primarily do so through functors between these categories. The insights given by functors will give us insights on how to translate subtypes between related types.
A functor F from a category $\mathcal{C}$ to a category $\mathcal{D}$ is a map such that:

- given $X \in Ob(\mathcal{C})$, $F(X) \in Ob(\mathcal{D})$
- F maps any morphism $f : X \to Y$ in $\mathcal{C}$ to a morphism $F(f) : F(X) \to F(Y)$ in $\mathcal{D}$
- $F(id_X) = id_{F(X)}$
- $F(g \circ f) = F(g) \circ F(f)$

One kind of functor we are interested in is a functor from $\mathcal{S}(int1)$ to $\mathcal{S}(int2)$, where both $int1$ and $int2$ are bounded integers.
Let Y be a bounded integer type with lower bound a and upper bound b and let X be a bounded integer type with lower bound $x < a$ and upper bound $y < b$. Let $\iota_{X\to Y} : \mathcal{S}(X) \to \mathcal{S}(Y)$ be a map that maps any subtype $A \in Ob(\mathcal{S}(X))$ to subtype $B \in Ob(\mathcal{S}(Y))$ where

$$\mathcal{P}_B = \mathcal{P}_A \land var \geq x \land var \leq y$$

(var being a placeholder name for the subtyped variable).
Let $\iota_{X\to Y}$ map morphisms

$$f : A \to B$$

to morphisms

$$\iota_{X\to Y}(f) : \iota_{X\to Y}(A) \to \iota_{X\to Y}(B)$$

For this to be well-defined, we need to show that such morphisms always exist in $\mathcal{S}(Y)$. Given $A, B, C \in Ob(\mathcal{S}(X))$ with morphisms

$$f : A \to B, g : B \to C$$

we find that the existence of f and g mean that $\mathcal{P}_A \Rightarrow \mathcal{P}_B$ and $\mathcal{P}_B \Rightarrow \mathcal{P}_C$. As such, we know that

$$\mathcal{P}_{\iota_{X\to Y}(A)} = \mathcal{P}_A \land var \geq x \land var \leq y$$
$$\mathcal{P}_{\iota_{X\to Y}(B)} = \mathcal{P}_B \land var \geq x \land var \leq y$$

and

$$(\mathcal{P}_A \land var \geq x \land var \leq y) \Rightarrow (\mathcal{P}_B \land var \geq x \land var \leq y)$$

As such, we know that $\iota_{X\to Y}(f)$ exists. For the same reason, we know that $\iota_{X\to Y}(g)$ exists. Because there is at most one morphism from $\iota_{X\to Y}(A)$ to $\iota_{X\to Y}(C)$, we know that

$$\iota_{X\to Y}(g \circ f) = \iota_{X\to Y}(g) \circ \iota_{X\to Y}(f)$$

Furthermore, we find that

$$\iota_{X\to Y}(id_X) = id_{\iota_{X\to Y}(X)}$$

As such $\iota_{X\to Y}$ is in fact a functor.

7 CONCLUSION

We have determined a set of rewrite rules for verifying predicate subtypes using VerCors. Alongside these rewrite rules, we have given the original motivation for these rules. This allows a guideline for extending the list of rewrite rules to support applying predicate subtypes in more contexts. Through this, we have created a template for adding predicate subtyping to VerCors in such a way that bounded integers may be supported by it.
Furthermore, we have created a prototype implementation to show that implementing this template is practically possible. This is only a partial implementation, since time constraints made implementing all planned features infeasible.
We have also described the intended semantics for predicate subtypes in a context-agnostic manner, providing a base for mathematical proofs about the predicate subtyping system.

Further work

As was mentioned in the implementation section, the prototype does not implement all planned functionality yet. As such, the prototype may be further expanded to support the unrealised features. In this thesis, we studied the semantics of predicate subtyping only with regard to classical first order logic. An investigation of the properties of our category with respect to different logics may highlight which parts of the semantics of the category are dependent on the logic used. Furthermore, we believe that it may be worth exploring how $\mathcal{S}(X)$ and boolean categories are related [15]. This suggestion is made because of similarities between the objects with universal properties in $\mathcal{S}(X)$ and those in a boolean category describing first order logic.
We have also not yet shown whether the $\mathcal{S}(X)$ categories for bounded integers form subcategories of $\mathcal{S}(Y)$ for bounded integers with bounds that include X. Further functors that may be of interest for further investigation are functors from interface subtypes to the subtypes of an implementation of said interface.

REFERENCES

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification, In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). *Proc. ACM Program. Lang.* 3, OOPSLA, 147:1–147:30. https://doi.org/10.1145/3360573

[2] William Babonnaud. 2021. Covariant Subtyping Applied to Semantic Predicate Calculi. In *LACL 2021 - Logical Aspects of Computational Linguistics*. Montpellier (online), France. https://inria.hal.science/hal-03542057

[3] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. [n. d.]. *WP Plug-in Manual*. CEA LIST, Software Safety Laboratory. https://frama-c.com/download/wp-manual-Chlorine-20180501.pdf

[4] Greta Coraglia and Jacopo Emmenegger. 2023. Categorical models of subtyping. *ArXiv* abs/2312.14600 (2023). https://api.semanticscholar.org/CorpusID:266520932

[5] Brendan Fong and David I. Spivak. 2019. *Resource Theories: Monoidal Preorders and Enrichment*. Cambridge University Press, 38–76.

[6] Philippe Herrmann and Julien Signoles. [n. d.]. *Frama-C's annotation generator plug-in*. CEA LIST, Software Safety Laboratory. https://frama-c.com/download/rte-manual-Sulfur-20171101.pdf

[7] Gabriel Hondet. 2022. *Expressing predicate subtyping in computational logical frameworks*. Université Paris-Saclay. https://theses.hal.science/tel-03855351/

[8] Joe Hurd. 2001. Predicate Subtyping with Predicate Sets. In *Theorem Proving in Higher Order Logics*, Richard J. Boulton and Paul B. Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–280.

[9] Joachim Lambek. 1974. Functional completeness of cartesian categories. *Annals of Mathematical Logic* 6, 3-4 (1974), 259–292.

[10] F. William Lawvere. 1969. Diagonal arguments and cartesian closed categories. In *Category Theory, Homology Theory and their Applications II*. Springer Berlin Heidelberg, Berlin, Heidelberg, 134–145.

[11] Gary Leavens, Albert L. Baker, and Clyde D. Ruby. 2006. Preliminary design of JML: a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes* 31 (2006), 38 pages. https://doi.org/10.1145/1127878

[12] Ursula Martin, Erik A. Mathiesen, and Paulo Oliva. 2006. Hoare Logic in the Abstract. In *Computer Science Logic*, Zoltán Ésik (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 501–515.

[13] prustiI32 [n. d.]. *Heap-based type encoding*. https://viperproject.github.io/prusti-dev/dev-guide/encoding/types-heap.html

[14] John Rushby, Sam Owre, and N. Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering* 24 (1998), 709–720. https://ieeexplore.ieee.org/abstract/document/713327

[15] Lutz Strassburger. 2006. What could a Boolean category be?. In *Classical Logic and Computation 2006 (ICALP Workshop)*. Venice, Italy. https://inria.hal.science/inria-00130504

[16] viperPreds [n. d.]. *Viper tutorial*. http://viper.ethz.ch/tutorial/#predicates