A Symbolic Representation for Wildcard Permissions in Viper

Bachelor's Thesis Project Description

Yanick Bachmann
yanickb@ethz.ch

Programming Methodology Group
Department of Computer Science
ETH Zürich

Supervisor:
Thibault Dardinier

February 12, 2021

1 Background

The Viper verification infrastructure [MSS15] provides an intermediate verification language and two back-end verifiers for this language. Various existing automatic verifiers for programming languages such as Python, Rust, Go, and Java verify programs by translation to Viper. At the heart of Viper is a permission model that is useful to reason about heap-manipulating programs and thread interactions in concurrent software. Viper uses permissions to express ownership of heap locations.

These are represented as fractions (rational numbers). That is, a permission to a heap location $x.f$ (where x is a reference and f a field) is a rational number p between 0 and 1. If $p = 0$ (no permission), then the program cannot read the value of $x.f$. If $p = 1$ (exclusive permission), then the program can read and modify $x.f$. However, exclusive permissions are often too restrictive. Typically, it is safe for multiple threads to concurrently access the same heap location, as long as they only read this heap location (and do not modify it). Therefore, when the permission p is such that $0 < p < 1$, the program can read the value of $x.f$, but not write it.

To allow specifying read permissions, Viper provides *wildcard* permissions. A wildcard permission is a permission amount that is unspecified, but known to be greater than zero. Viper currently automates it as an existential fraction. That is, a Viper program owns a wildcard permission to $x.f$ if and only if there exists a fraction $p > 0$ such that the program has at least permission p to $x.f$. The wildcard permission amount provides a convenient way to encode read-only resources that can be shared among threads.

2 Our Approach

The aim of this project is to explore and implement a different representation for wildcards, inspired by the VeriFast verifier [JSP+11] (a verifier for C and Java programs), that we refer to as *symbolic* wildcards and describe below. The motivation for this project is twofold. First, we want to understand how this representation can be integrated with other advanced Viper features. Secondly, we expect that a symbolic encoding of wildcards will be more reliable and yield better performances than the current encoding with existential permissions, mainly because it will not have to deal with arithmetic.

The VeriFast verifier represents wildcard permissions (referred to as *dummy fractions*) symbolically. Exhaling (giving away) a dummy fraction in VeriFast amounts to converting a concrete fractional permission to a dummy fraction (which is infinitely duplicable). Thus, once a dummy fraction has been given away, write permission cannot be regained.

Such an implementation of wildcards seems more natural as the following example shows. It is possible to hold a fraction of a predicate in Viper, for example to allow several threads to concurrently read the same data structure. When a fraction of a predicate is unfolded, all permissions it contains are multiplied by the corresponding fraction, as we illustrate on Listing 1. In this example, unfolding a wildcard amount of the read predicate should result in a multiplication of two wildcard permissions. To avoid non-linear arithmetic issues with the underlying SMT solver, the Viper back-end verifier Carbon simplifies this multiplication to a wildcard permission. Such an ad hoc optimization will not be necessary with symbolic wildcards, where the multiplication of a symbolic wildcard with any non-zero permission simply results in a symbolic wildcard.

Listing 1: Example of permission multiplication for predicates

```
0  field f: Int
1
2  predicate read(x: Ref)
3  {
4      acc(x.f, wildcard)
5  }
6
7  method main(x: Ref)
8      requires acc(read(x), wildcard)
9  {
10     // This unfold multiplies two wildcard permissions,
11     // resulting in another wildcard permission to x.f,
12     // which allows reading the value of x.f
13     unfold acc(read(x), wildcard)
14     var x: Int := x.f
15 }
```

One caveat symbolic wildcards bring with them, as written above, is that write permission cannot be regained once a wildcard has been exhaled. On the other hand, Viper currently represents wildcards as existential fractions, which makes it possible to inspect the fractional permission owned, and thus to regain write permission to a heap location after a wildcard permission amount has been exhaled. Listing 2 illustrates this. The program state begins with write permission, and then exhales a wildcard permission amount. Permission introspection (`perm(x.f)`) allows the currently held permission amount to be stored into the variable `p`. Using this variable we can calculate the missing permission amount we require to be able to write the heap location, and inhale it, which is something one cannot achieve in VeriFast. It is not clear, however, how useful this feature is in practice.

Listing 2: Example wildcard permission reversal

```
0  field f: Int
1
2  method recover_wildcard(x: Ref)
3  requires acc(x.f)
4  {
5      // remove a wildcard permission amount
6      exhale acc(x.f, wildcard)
7      // Permission introspection
8      var p: Perm := perm(x.f)
9      // regain full write permission
10     inhale acc(x.f, write - p)
11     // write to the heap location
12     x.f := 5
13 }
```

With this in mind, we will implement symbolic wildcards by modifying the Viper back-end Carbon, which translates Viper programs to Boogie [Lei08] programs, which are then verified using an SMT solver. We will first consider a naive implementation of wildcard permissions that uses a map from heap locations to Boolean values that encode the property of holding a wildcard. Reading from a heap location is permitted if either a fractional permission $p : 0 < p$ or a symbolic wildcard is held for this location.

Our naive implementation could result in an exponential blowup of the number of branches the SMT solver has to explore, because of the use of disjunctions to check heap accessibility (each disjunction potentially multiplies by two the number of branches to explore). We will take special care to avoid such issues, since our aim is to find a viable encoding of symbolic wildcards that satisfies both performance and reliability. Nonetheless, symbolically representing wildcards could yield a performance increase since the underlying SMT solver would not need to work with existential fractions but Boolean values, and thus would avoid arithmetic.

A further challenge will be the exploration of the interaction of the new wildcard representation with other Viper features such as predicates folding and unfolding, quantified permissions, or known-folded permissions [HKMS13]. These are key for this representation to be usable in practice.

3 Core Goals

The goal of this thesis is to explore and implement a symbolic representation of wildcard permissions for Viper. The core milestones of this project are:

1. Explore and formally define a semantics for this symbolic representation of wildcard permissions.
2. Implement symbolic wildcards in the Viper back-end verifier Carbon for a subset of Viper. This subset should include predicate (un)folding, quantified permissions, and known-folded permissions.
3. Evaluate the implementation of symbolic wildcards in terms of expressiveness, performance, and reliability, by comparing it to the current implementation of wildcards.

4 Extension Goals

1. Extend the implementation to other advanced features, such as magic wands or permission introspection.
2. Extend the implementation of symbolic wildcards to do permission counting [BCOP05].
3. Explore a reverse conversion of symbolic wildcards back to concrete fractions.

Bibliography

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 259–270, New York, NY, USA, 2005. Association for Computing Machinery.

[HKMS13] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *ECOOP, pages 451–476*, 2013.

[JSP+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods (NFM). pp. 41–55*, 2011.

[Lei08] K. Rustan M. Leino. This is boogie 2. In *Microsoft Research, Redmond, WA, USA*, 2008.

[MSS15] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation pp 41-62*, 2015.