Viper

A Verification Infrastructure for Permission-Based Reasoning

Uri Juhasz, Ioannis Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, Alex Summers (and several students)

1st March 2015, ECOOP’15 PC Meeting, Zurich
Automatic Program Verification

- Safety (memory accesses, non-null, ...)
- Correctness (functional specs)
- Termination, message replies, ...
Verification using Automatic Provers

- Automatic first-order logic tools – major progress in the last decade (SAT, SMT)
- Intermediate verification languages - Boogie, Why, ...
- Back-end: verifier
  (verification condition generator)

= Common infrastructure for building front-end verifiers
Verification using Automatic Provers

- Automatic first-order logic tools – major progress in the last decade (SAT, SMT)

- Intermediate verification languages - Boogie, Why, ...

- Back-ends: verifiers - but also inference engines, slicers, static analysers, ...

= Common infrastructure for building front-end verifiers
Verification using Automatic Provers

Common infrastructure enabled many *success stories* and *tools*

- Microsoft Hypervisor (VCC)
- Device drivers (Corral)
- Spec#, Dafny
- Krakatoa, Jessie
- Frama-C, Why3
- ...

Front-ends

Back-ends

Prog. languages + specifications

Automatic provers

IVL
Permission-Based Reasoning

Separation logic and others *permission logics*:

- *Locally* reason about *shared mutable state*
- Many *successful applications*, including
  - Device driver safety (Microsoft)
  - Belgian Electronic Identity Card
- Many on *going developments*
  (esp. fine-grained concurrency)

Not a first-order logic
→ Significantly complicates using existing provers
Permission-Based Reasoning

Consequence: many *custom verification engines* (usually based on symbolic execution): Smallfoot, VeriFast, jStar, ...

Alternative: Encoding SL into FOL (e.g. Chalice)
Viper: Our Verification Infrastructure

Silver:
- *Native* support for permissions
- Few (but expressive) constructs
- Designed with verification and inference in mind

Back-ends: Two verifiers; plans to develop inference, slicer

Front-ends (proof of concept):
- *Chalice* (concurrency research)
- *Scala* (very small subset)
- *Java* (VerCors, U Twente)
- *OpenCL* (VerCors, U Twente)
Modular Static Verification + Shared State

\[
\text{foo}(x) \quad \text{bar}(x)
\]
Modular Static Verification + Shared State

foo(x)  bar(x)
Modular Static Verification + Shared State

foo(x)  bar(x)
Permissions

foo(x)  bar(x)
Permission Transfer

$\text{foo}(x)$  $\text{bar}(x)$

[Diagram showing the transfer of permissions with question marks and check marks.]
Permission Transfer

foo(x)  bar(x)
Fractional Permissions

foo(x)  bar(x)
Splitting Fractional Permissions

$\text{foo}(x) \quad \text{bar}(x)$
Merging Fractional Permissions

foo(x)  bar(x)
Permission Transfer

Idea of *permission transfer* generalises
- Fork-join (transfer between threads)
- Locks (transfer to/from lock invariant)
- Message passing (pass permissions)

Common operations
- **Gain** permissions
- **Lose** permissions
Silver: Inhale and Exhale Statements

Statement **exhale** $A$ means
- Assert and remove permissions required by $A$
- Assert logical constraints in $A$ (e.g. $c \cdot f = 0$)
- Havoc locations to which all permissions is lost (i.e. forget their values)

Statement **inhale** $A$ means
- Gain permissions required by $A$
- Assume logical constraints in $A$
Silver: Assertion Language Basics

Based on *implicit dynamic frames*

**Accessibility predicates** denote permissions

**Assertions may be heap-dependent**

**Fractional permissions**

Conjunction *sums up* permissions (similar to * in separation logic)
Demo
Silver: Language Features

Objects and fields, if-then-else, methods (with pre/post specs), loops (with invariants)

No notion of concurrency (encode via inhale/exhale)

Simple type system
  – Int, Bool, Ref, Perm
  – Mathematical sets Set[T] and sequences Seq[T]
Unbounded data structures via *recursive predicates*

```plaintext
predicate list(x: Ref) {
    acc(x.val) && acc(x.next)
    && (x.next != null ==> list(x.next))
}
```

*fold/unfold* statements exchange predicate instances for their bodies (not automatic due to recursion)

*Heap-dependent, pure abstraction functions*

```plaintext
function elems(x: Ref): Seq[Int]
    requires list(x)
{ unfolding list(x) in
    [x.val] ++ (x.next == null ? [] : elems(x.next))
}
```
**Silver: Custom Mathematical Domains**

*Domains* to specify custom mathematical types

- Type-parametric domains
- Domain functions
- Domain axioms

```plaintext
domain Pair[X,Y] {
  function pair(x: X, y: Y): Pair[X,Y]
  function first(p: Pair[X,Y]): X

  axiom forall x: X, y: Y • first(pair(x,y)) == x
}
...
method foo(x: Ref, p: Pair[Int, Int])
  requires acc(x.f)
  { x.f := first(p) }
```
Abstract read permissions

- Alternative to fractional permissions
- No need to commit to concrete fractions, e.g. $\frac{1}{2}$

```
method foo(x: Ref, p: Perm)
  requires 0 < p && acc(x.f, p)
{
  // read x.f
  if (*) {
    var q: Ref
    constraining (q) {
      foo(x, q) // give away q < p
    }
  }
}
```

Allows unbounded splitting and counting
Paired assertions \([A, B]\)
- When inhale, A is used
- When exhaled, B is used
- Asymmetry justified elsewhere
(type system, soundness proof, proof principle, ...)

\[
\begin{align*}
\forall x : \text{Nat} \cdot P(x), \\
\forall x : \text{Nat} \cdot
\begin{cases}
\forall y : \text{Nat} \cdot y < x \implies P(y) & \text{implies} \\
P(x)
\end{cases}
\end{align*}
\]
http://bitbucket.org/viperproject/

- **Chalice**
- **Scala**
- **Java (U Twente)**
- **OpenCL (U Twente)**

**Silver AST**
- gene rate
- infer additional specifications
- verified by
- encodes in
- queries
- queries

**Static Analysis**

**Carbon**

**Silicon**

**Boogie (Microsoft)**

**Z3 (Microsoft)**