Prototyping Relaxed Separation Logic in Viper

Peter Müller
Viper
Verification
condition
generation
Symbolic
execution
Abstract
interpretation
Boogie
SMT solver
Separation Logic

- Heap properties are specified via points-to assertions

  \[ x.f \rightarrow v \]

- Each heap access to x.f requires permission to x.f

  \[ \{ x.f \rightarrow _ \} \ x.f := v \ \{ x.f \rightarrow v \} \]
Separating Conjunction

- Composition of heaps is described using separating conjunction

\[ \{ P \} \ S \ { Q \} \]
\[ \{ P \ast R \} \ S \ { Q \ast R \} \]

- Frame rule

\[ \{ P \} \ S \ { Q \} \]
\[ \{ P \ast R \} \ S \ { Q \ast R \} \]

- Parallel composition

\[ \{ P_1 \} \ S_1 \ { Q_1 \} \]
\[ \{ P_2 \} \ S_2 \ { Q_2 \} \]
\[ \{ P_1 \ast P_2 \} \ S_1 \parallel S_2 \ { Q_1 \ast Q_2 \} \]
Permission Transfer

\[
\begin{align*}
\{P\} & \text{ method } m \{Q\} \\
\{P\} & \text{ e.m()} \{Q\} \\
\{P \ast R\} & \text{ e.m()} \{Q \ast R\}
\end{align*}
\]

\[
\begin{align*}
\{P_1\} & S_1 \{Q_1\} \quad \{P_2\} & S_2 \{Q_2\} \\
\{P_1 \ast P_2\} & S_1 \parallel S_2 \{Q_1 \ast Q_2\} \\
\{P_1 \ast P_2 \ast R\} & S_1 \parallel S_2 \{Q_1 \ast Q_2 \ast R\}
\end{align*}
\]
Exhale and Inhale

**exhale** $P$
- Assert all logical constraints in $P$
- Check and remove permissions described by $P$

**inhale** $Q$
- Obtain permissions described by $Q$
- Assume all logical constraints in $Q$

- Analogs of **assert** and **assume**
Encoding Monitors

class Account {
    var bal: int

    invariant this.bal → _

    method deposit(amount: int)
    {
        acquire this
        this.bal := this.bal + amount
        release this
    }
}
Weak Memory

- Modern hardware often does not provide sequentially consistent shared memory
- Weak memory permits behaviors that are not possible under sequential consistency
- However, data-race free programs have only sequentially consistent behaviors

Possible results:
under SC: 10, 01, 11
under WM: also 00
C11

- The C11 memory model provides several kinds of variables

- Non-atomic variables
  - Data races are errors

- Atomic variables with release-write and acquire-read
  - Writes and reads are synchronized

- Relaxed separation logic (RSL) supports some features of the C11 memory model

```c
n = alloc_{NA}(0);
a = alloc_{RA}(0);
n = 5;
if(a.load() == 1)
a.store(1);
assert n == 5;
```
Non-Atomic Variables

- Permissions prevent data races on non-atomic variables

\[
\{ \text{true} \} \ x = \text{alloc}_{\text{NA}}(v) \ \{ x \rightarrow v \}
\]

\[
\{ x \rightarrow _{} \} \ \*x = v \ \{ x \rightarrow v \}
\]

\[
\{ x \rightarrow V \} \ t = \*x \ \{ x \rightarrow V \* t = V \}
\]
Release-Acquire

- Races on atomic variables are permitted
- Release-acquire can be seen as message passing
- Messages may transfer permissions to non-atomic variables

n = alloc_{NA}(0);
a = alloc_{RA}(0);
n = 5;
if(a.load() == 1)
a.store(1);
assert n == 5;
Location Invariants

- Location invariant $Q(v)$ specifies an assertion that holds when the location has value $v$

- Acquire-read of value $v$ transfers permissions of $Q(v)$ from atomic variable to thread

- Release-write of value $v$ transfers permissions of $Q(v)$ from thread to atomic variable

$$Q(v) \equiv \begin{cases} n \rightarrow 5 & \text{if } v = 1 \\ \text{true} & \text{otherwise} \end{cases}$$
Proof Rules

- Choose location invariant when allocating an atomic location

\[
\{ Q(v) \} \ x = \text{alloc}_{RA}(v) \ \{ \text{Rel}_Q(x) \ast \text{Acq}_Q(x) \}
\]

- Release-write gives up permissions

\[
\{ \text{Rel}_Q(x) \ast Q(v) \} \ x.\text{store}(v) \ \{ \text{Rel}_Q(x) \}
\]

- Acquire-read gains permissions

\[
\{ \text{Acq}_Q(x) \} \ t = x.\text{load}() \ \{ Q(t) \ast \text{Acq}_Q(x) \}
\]
Proof Rules

- Reading the same value more than once would duplicate permissions

\[ \{ \text{Acq}_Q(x) \} \; t = x.\text{load()} \; \{ Q(t) \; \ast \; \text{Acq}_{Q[t := \text{true}]}(x) \} \]

\[ Q(v) = \begin{cases} n \rightarrow 5 & \text{if } v = 1 \\ \text{true} & \text{otherwise} \end{cases} \]

\[ x = a.\text{load}(); \]
\[ y = a.\text{load}(); \]
\[ \text{if}(x == y) \]
\[ \text{assert} \; \text{false}; \]
Proof Outline

\[
Q(v) \equiv \begin{cases} 
  n \rightarrow 5 & \text{if } v = 1 \\
  \text{true} & \text{otherwise}
\end{cases}
\]

\[
\begin{align*}
\{ \text{true} \} \\
n &= \text{alloc}_{NA}(0); \\
\{ n \rightarrow 0 \} \\
a &= \text{alloc}_{RA}(0); \\
\{ n \rightarrow 0 \ast \text{Rel}_Q(a) \ast \text{Acq}_Q(a) \}
\end{align*}
\]

\[
\begin{align*}
\{ n \rightarrow 0 \ast \text{Rel}_Q(a) \} \\
n &= 5; \\
\{ n \rightarrow 5 \ast \text{Rel}_Q(a) \} \\
a.\text{store}(1); \\
\{ \text{Rel}_Q(a) \}
\end{align*}
\]

\[
\begin{align*}
\{ \text{Acq}_Q(a) \} \\
\text{if}(a.\text{load()} == 1) \\
\{ Q(1) \ast \text{Acq}_Q(a) \} \\
\text{assert } n == 5;
\end{align*}
\]
Intermediate languages enable prototyping and reuse of infrastructure

Permissions enable framing and reasoning about concurrency

RSL proof rules can be encoded into Viper

n = alloc_{NA}(0);

a = alloc_{RA}(0);

n = 5;

if(a.load() == 1)

a.store(1);

assert n == 5;

\{ P \} S \{ Q \}

\{ P \ast R \} S \{ Q \ast R \}

\{ P_1 \} S_1 \{ Q_1 \} \{ P_2 \} S_2 \{ Q_2 \}

\{ P_1 \ast P_2 \} S_1 \parallel S_2 \{ Q_1 \ast Q_2 \}

viper.ethz.ch