Separation Logic
Faulty Logic
and Monads

Yin Wang

A Survey Project (sort of)

- Motivated by highly suspicious connections between Separation Logic, ~~Faulty Logic~~, and monads
- Reading list of 20+ papers
- Bored by Faulty Logic because stopped worrying about high energy particles from space
- Discovered connections between Separation Logic and ST monad
- Discovered the essence of Separation Logic (hopefully)

Recap of Separation Logic

[Reynolds 2002]

$$
\begin{array}{lll}
\langle \text{assert} \rangle ::= & \cdots & \\
& \mid \ \mathbf{emp} & \text{empty heap} \\
& \mid \ \langle \text{exp} \rangle \mapsto \langle \text{exp} \rangle & \text{singleton heap} \\
& \mid \ \langle \text{assert} \rangle \ * \ \langle \text{assert} \rangle & \text{separating conjunction} \\
& \mid \ \langle \text{assert} \rangle \ -\!\!* \ \langle \text{assert} \rangle & \text{separating implication}
\end{array}
$$

Inference Rules

Allocation (backwards reasoning)

$$\frac{}{\{\forall v'.\ (v' \mapsto \overline{e}) \mathrel{-\!\!*} p'\}\ v := \mathbf{cons}(\overline{e})\ \{p\},}$$

Mutation (backwards reasoning)

$$\frac{}{\{(e \mapsto -)\ *\ ((e \mapsto e') \mathrel{-\!\!*} p)\}\ [e] := e'\ \{p\}.}$$

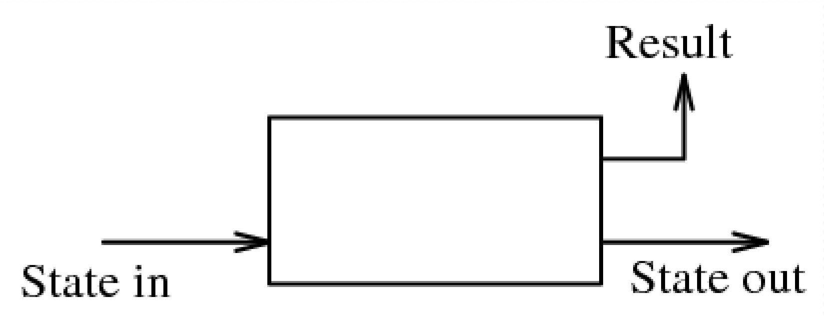
Lookup (alternative backward reasoning)

$$\frac{}{\{\exists v'.\ (e \hookrightarrow v') \wedge p'\}\ v := [e]\ \{p\},}$$

ST Monad

[Launchbury 1994]

- (ST s a) is a computation which transforms a state "indexed by type s"



State Transformers

```
newVar   :: a -> ST s (MutVar s a)
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

Composing Transformers (bind)

```
thenST :: ST s a -> (a -> ST s b) -> ST s b
```

State Transformers

Mutable reference indexed by “type s”

```
newVar   :: a -> ST s (MutVar s a)
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

Composing Transformers (bind)

```
thenST :: ST s a -> (a -> ST s b) -> ST s b
```

State Transformers

Mutable reference indexed by “type s”

```
newVar   :: a -> ST s (MutVar s a)
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

Composing Transformers (bind)

```
thenST :: ST s a -> (a -> ST s b) -> ST s b
```

State Transformers

Mutable reference indexed by “type s”

```
newVar   :: a -> ST s (MutVar s a)
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

Composing Transformers (bind)

```
thenST :: ST s a -> (a -> ST s b) -> ST s b
```

Encapsulation

```
runST :: ∀a. (∀s. ST s a) -> a
```

- rank-2 polymorphism (similar to System F) to encapsulate state with help from type system
- ensures “single-threaded access” *statically*

Encapsulation

```
runST :: ∀a. (∀s. ST s a) -> a
```

- rank-2 polymorphism (similar to System F) to encapsulate state with help from type system
- ensures "single-threaded access" *statically*

Does Separation Logic Support Multi-threaded access?

- “By using the frame rule, one can extend a local specification, involving the variable and *parts of the heap* that are actually used by c. Thus the frame rule is the key to *local reasoning* about the heap.”

Frame Rule

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}},$$

"Local reasoning": A feature or a restriction?

Frame Rule

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\},}$$

- "We can talk about other parts of the heap, as long as they are disjoint from what we talk about here."
- "We can talk about the same heap part only *once*."

Implementation of ST monad

- “The *state* of each encapsulated state thread is represented by a collection of objects in heap-allocated storage.”

The Connection?

- The Frame Rule and runST have the same purpose: to ensure *single-threaded heap access*.
- Neither ST monad nor Separation Logic support multi-threaded heap access.
- Separation Logic *talks* about the heap locally
- ST monad *uses* the heap locally

The Connection?

- The Frame Rule and runST have the same purpose: to ensure *single-threaded heap access*.
- Neither ST monad nor Separation Logic support multi-threaded heap access.
- Separation Logic *talks* about the heap locally
- ST monad *uses* the heap locally

Can ST monad “talk” too?

Model Separation Logic with ST monad

- Original plan: ST monad with state indexed by Separation Logic formulas
- Use thenST (bind) to do the inference
- Abandoned because a better way is found!

The Essence of Separation Logic

- Verification Condition generation:

$\{\forall v'.(v' \mapsto 42 \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_2) \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_1) \mathrel{-\!\!*} p_1))))\}$

$e := \mathrm{cons}(42)$

$\{(e \mapsto -) * ((e \mapsto e_2) \mathrel{-\!\!*} ((e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)))\}$

$[e] := e_2$

$\{(e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)\}$

$[e] := e_1$

$\{p_1\}$

The Essence of Separation Logic

- Verification Condition generation:

$\{\forall v'.(v' \mapsto 42 \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_2) \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_1) \mathrel{-\!\!*} p_1)))))\}$

$e := \mathrm{cons}(42)$

$\{(e \mapsto -) * ((e \mapsto e_2) \mathrel{-\!\!*} ((e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)))\}$

$[e] := e_2$

$\{(e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)\}$

$[e] := e_1$

$\{p_1\}$

The Essence of Separation Logic

- Verification Condition generation:

$\{\forall v'.(v' \mapsto 42 \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_2) \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_1) \mathrel{-\!\!*} p_1)))))\}$

$e := \mathrm{cons}(42)$

$\{(e \mapsto -) * ((e \mapsto e_2) \mathrel{-\!\!*} ((e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)))\}$

$[e] := e_2$

$\{(e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)\}$

$[e] := e_1$

$\{p_1\}$

The Essence of Separation Logic

- Verification Condition generation:

$$\{\forall v'.\,(v' \mapsto 42 \mathrel{-\!\!*} ((v' \mapsto -\,) * ((v' \mapsto e_2) \mathrel{-\!\!*} ((v' \mapsto -\,) * ((v' \mapsto e_1) \mathrel{-\!\!*} p_1))))\}$$

$$e := \mathrm{cons}(42)$$

$$\{(e \mapsto -\,) * ((e \mapsto e_2) \mathrel{-\!\!*} ((e \mapsto -\,) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)))\}$$

$$[e] := e_2$$

$$\{(e \mapsto -\,) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)\}$$

$$[e] := e_1$$

$$\{p_1\}$$

The Essence of Separation Logic

- Verification Condition generation:

$$\{\forall v'.\,(v' \mapsto 42 \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_2) \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_1) \mathrel{-\!\!*} p_1)))))\}$$

$$e := \mathrm{cons}(42)$$

$$\{(e \mapsto -) * ((e \mapsto e_2) \mathrel{-\!\!*} ((e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)))\}$$

$$[e] := e_2$$

$$\{(e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)\}$$

$$[e] := e_1$$

$$\{p_1\}$$

The Essence of Separation Logic

freshly bound (eigen) variable

- Verification Condition generation:

$$\{\forall v'.(v' \mapsto 42 \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_2) \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_1) \mathrel{-\!\!*} p_1))))\}$$

$$e := \mathrm{cons}(42)$$

$$\{(e \mapsto -) * ((e \mapsto e_2) \mathrel{-\!\!*} ((e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)))\}$$

$$[e] := e_2$$

$$\{(e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)\}$$

$$[e] := e_1$$

$$\{p_1\}$$

The Essence of Separation Logic

- Verification Condition generation:

$$\{(v' \mapsto 42) * ((v' \mapsto e_2) \mathrel{-\!\!*} ((v' \mapsto -) * ((v' \mapsto e_1) \mathrel{-\!\!*} p_1))\}$$

$$e := \mathrm{cons}(42)$$

$$\{(e \mapsto -) * ((e \mapsto e_2) \mathrel{-\!\!*} ((e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)))\}$$

$$[e] := e_2$$

$$\{(e \mapsto -) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)\}$$

$$[e] := e_1$$

$$\{p_1\}$$

The Essence of Separation Logic

- Verification Condition generation:

$$\{(v' \mapsto e_2) * ((v' \mapsto e_1) -\!\!* \, p_1))\}$$

$$e := \mathrm{cons}(42)$$

$$\{(e \mapsto -) * ((e \mapsto e_2) -\!\!* \, ((e \mapsto -) * ((e \mapsto e_1) -\!\!* \, p_1)))\}$$

$$[e] := e_2$$

$$\{(e \mapsto -) * ((e \mapsto e_1) -\!\!* \, p_1)\}$$

$$[e] := e_1$$

$$\{p_1\}$$

The Essence of Separation Logic

- Verification Condition generation:

$$\{p_1\}$$

$$e := \mathrm{cons}(42)$$

$$\{(e \mapsto -\ ) * ((e \mapsto e_2) \mathrel{-\!\!*} ((e \mapsto -\ ) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)))\}$$

$$[e] := e_2$$

$$\{(e \mapsto -\ ) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)\}$$

$$[e] := e_1$$

$$\{p_1\}$$

The Essence of Separation Logic

- Verification Condition generation:

$$\{p_1\}$$

$$e := \mathrm{cons}(42)$$

$$\{(e \mapsto -\,) * ((e \mapsto e_2) \mathrel{-\!\!*} ((e \mapsto -\,) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)))\}$$

$$[e] := e_2$$

$$\{(e \mapsto -\,) * ((e \mapsto e_1) \mathrel{-\!\!*} p_1)\}$$

$$[e] := e_1$$

$$\{p_1\}$$

separating
implication =
continuation?

Not Quite Continuation

- The "premise" of separating implication can be anywhere in the heap
- *Nondeterministic* continuation
- PSPACE complexity

Symbolic Execution vs Separation Logic

- So we see that the extracted Separation Logic precondition resembles a “program” with
  - *nondeterministic control flow*
  - variables in “*SSA form*”
- Seems to be redundant work. Can we do without extraction of Separation Logic formulas?

What about Branching?

$$\{\forall v'.(v' \mapsto 42 \mathrel{-\!\!*} (p \wedge ((v' \mapsto -\ ) * ((v' \mapsto e_1) \mathrel{-\!\!*} p_2)))$$
$$\vee (\neg p \wedge ((v' \mapsto -\ ) * ((v' \mapsto e_2) \mathrel{-\!\!*} p_2)))\}$$

$e := \mathrm{cons}(42)$

$(p \wedge ((e \mapsto -\ ) * ((e \mapsto e_1) \mathrel{-\!\!*} p_2))) \vee (\neg p \wedge ((e \mapsto -\ ) * ((e \mapsto e_2) \mathrel{-\!\!*} p_2)))$

$\mathrm{if}\ (p_1)\ \mathrm{then}$

$\{p \wedge ((e \mapsto -\ ) * ((e \mapsto e_1) \mathrel{-\!\!*} p_2))\}$

$[e] := e_1$

else

$\{\neg p \wedge ((e \mapsto -\ ) * ((e \mapsto e_2) \mathrel{-\!\!*} p_2))\}$

$[e] := e_2$

$\{p_2\}$

Strategies for Branches

- Conditions should be put into control-flow if cannot be determined statically, hoping it will be eliminated by resolution.
- State :: Disj(Conj Formula, (Stack, Heap))
- Loop invariants can be treated similar way (put invariants into control flow)

Prototype Implementation

- An interpreter to produce the state (primitive status)
- A theorem prover that can handle nondeterminism for heaps (TODO)

Is This Model Checking?

- Model Checking is for verifying that a *specific* model satisfies the specification.
- The "model" generated by the interpreter is NOT only a specific model.
- It is a "WLOG model", because it prohibit access to the underlying representation of the heap and addresses.
- This model does not entail extra information.
- Every heap cell can be thought of an "atomic formula" without names

Relation to Separation Logic

- Symbolic execution says what Separation Logic can possibly “say” as the *post-condition*.
- Unlike Separation Logic, it doesn’t say it until it is forced to speak.
- A theorem prover must be used to force the model to speak.
- Likely to be undecidable with quantifiers (same limitation as Separation Logic.)

Future Directions

- Implement the theorem prover which can handle nondeterministic heap as input.
- Possibly need to change the interpreter to be reversible (rewrite in Coq?).
- Why do I want to do this after all? Separation Logic is a very impractical tool. C/C++ already have symbolic execution tools (e.g. clang checker).
- But I learned a lot about state, threads and concurrency.

Thank you!

- Questions?