Hoare-like Logics for Verifying and Inferring Conditional Information Flow

Torben Amtoft & Anindya Banerjee & John Hatcliff & Edwin Rodríguez & Joey Dodds & ...

Kansas State University

19th CREST Open workshop, May 1, 2012

Dependency and Non-Interference

Consider command C

$z := x + y$

Dependency perspective:

the value of z after executing C depends only on (at most) x,y

Non-interference perspective:

if two stores agree on x,y before C then they will agree on z after C

Expressed as triple in Hoare-like logic:

$$\{x\ltimes,\ y\ltimes\}\ C\ \{z\ltimes\}$$

where $\ltimes$ introduces a two-store assertion:

$s_1 \& s_2 \models E\ltimes$ *iff* $[\![E]\!]_{s_1} = [\![E]\!]_{s_2}$

Semantics of Hoare Triples

A Hoare-triple $\{\Theta\}\ C\ \{\Theta'\}$ with 2-assertions denotes:

if $s_1 \& s_2 \models \Theta$
and $s_1\ [\![C]\!]\ s_1'$
and $s_2\ [\![C]\!]\ s_2'$
then $s_1' \& s_2' \models \Theta'$

This is termination-insensitive:

- if C loops on s_1 and/or on s_2
- then correctness holds vacuously.

To get termination sensitivity, one might introduce $\perp\!\ltimes$:

$$\{x\ltimes\}\ C\ \{\perp\!\ltimes\}$$

would then say that if $s_1(x) = s_2(x)$ then either

1. C terminates on s_1 and on s_2, or
2. C loops on s_1 and on s_2

Conditional Dependencies

Now consider command

$$\textbf{if } B \textbf{ then } z := x \textbf{ else } z := y$$

In terms of noninterference: two stores will end up agreeing on z if they

1. agree on B
2. agree on x when B is true
3. agree on y when B is false

This may be expressed as the 2-assertion Hoare triple:

$$\{B\ltimes,\ B \Rightarrow x\ltimes,\ \neg B \Rightarrow y\ltimes\}\ C\ \{z\ltimes\}$$

Semantics of a conditional assertion:

$$s_1 \& s_2 \models \phi \Rightarrow E\ltimes \text{ iff } s_1 \models \phi,\ s_2 \models \phi \text{ implies } [\![E]\!]_{s_1} = [\![E]\!]_{s_2}$$

Inference Algorithm

Goal:

1. given command
2. given postcondition (often unconditional)
3. infer precondition that yields correct Hoare triple

Applications:

- derive (procedure) contracts
- check user-supplied contract:
  does given precondition entail inferred precondition?

The inferred precondition is not necessarily the weakest:

- loops are approximated
- for procedures, summaries are consulted
- ...?

Analyzing Assignments

For assignment $x := E$, as in standard Hoare Logic, the (weakest) precondition is found by substituting E for x in postcondition

$$
\begin{array}{lcc}
\Theta & y + z > 7 \Rightarrow w \ltimes & w > 5 \Rightarrow (y + z) \ltimes \\
 & \uparrow & \uparrow \\
 & \multicolumn{2}{c}{x := y + z} \\
 & | & | \\
\Theta' & x > 7 \Rightarrow w \ltimes & w > 5 \Rightarrow x \ltimes
\end{array}
$$

Special Case: Conclusion Not Modified

When C does not modify z, consider the triple

$$\{\phi \Rightarrow z\ltimes\}\ C\ \{\phi' \Rightarrow z\ltimes\}$$

For this to be valid, it must hold that:

- if post-stores are forced to agree on z
- then also pre-stores must be forced to agree on z

which amounts to ϕ satisfying

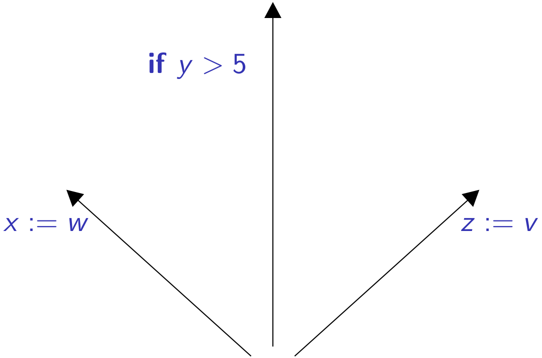
$$\forall s, s' : \mathit{if}\ s\ [\![C]\!]\ s'\ \mathit{and}\ s' \models \phi'\ \mathit{then}\ s \models \phi$$

This kind of resembles saying $\phi = wp(C, \phi')$

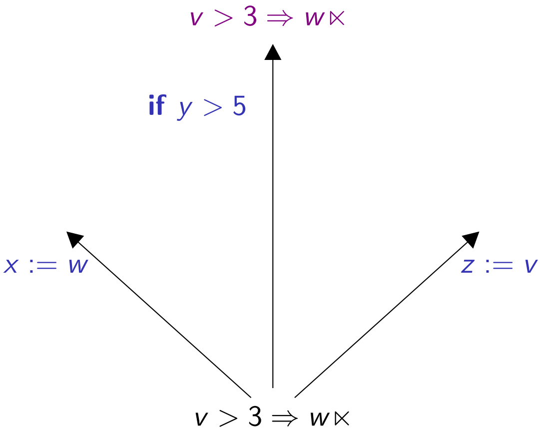
- but the direction is backwards
- and approximation is upwards: $\phi = \mathit{true}$ is safe

We call this Necessary PreCondition (NPC)

Analyzing Conditionals



Analyzing Conditionals



Analyzing Conditionals

$$(z > 3 \land y > 5) \lor (v > 3 \land y \leq 5) \Rightarrow w\ltimes$$

if $y > 5$

$x := w$

$z := v$

$$z > 3 \Rightarrow w\ltimes$$

Analyzing Conditionals

$z > 7 \wedge y > 5 \Rightarrow w \ltimes \qquad v > 7 \wedge y \leq 5 \Rightarrow x \ltimes$

$(z > 7 \wedge y > 5) \vee (v > 7 \wedge y \leq 5) \Rightarrow (y > 5) \ltimes$

if $y > 5$

$z > 7 \Rightarrow w \ltimes \qquad v > 7 \Rightarrow x \ltimes$

$x := w \qquad z := v$

$z > 7 \Rightarrow x \ltimes$

Context

- Part of our work was motivated by a larger industrial collaboration effort with Rockwell Collins
- Rockwell Collins is developing multiple product lines of embedded information security devices following the MILS architecture
- Code size is relatively small (3-5K LOC) and confined to a particular style: a lot of buffer processing, copying, filtering
- These products must be certified and secure information flow and separation policies are primary concerns
- Each of these products has critical subsystems code in SPARK, a safety-critical subset of Ada that is suitable for formal reasoning (no heap)
- SPARK information flow contracts are being used to support certification cases

Information Flow Contracts

SPARK provides "information flow" contracts that describe how a procedure causes information to flow from one variable to another


```
procedure Operate;
--# global out KeyStore.RotorValue,Encrypted;
--# in out KeyStore.SymmetricKey;
--# in Clear;
```

"out"-only variables are not read

"in/out"- both read and written

"in"-only variables are not written

Start by identifying procedure inputs and outputs (both parameters and any globals used) -- provides "frame conditions"

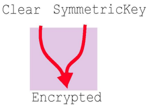
"Enforcing Security and Safety Models with an Information Flow Analysis Tool"

Information Flow Contracts

SPARK provides "information flow" contracts that describe how the associated procedures causes information to flow from one variable to another


```
procedure Operate;
--# global out KeyStore.RotorValue,Encrypted;
--# in out KeyStore.SymmetricKey;
--# in Clear;
--# derives
--#   KeyStore.SymmetricKey, KeyStore.RotorValue
--# from
--#   KeyStore.SymmetricKey
--# &
--#   Encrypted
--# from
--#   Clear, KeyStore.SymmetricKey
--# ;
```

Spark information flow annotations...



Information flows from `Clear`, `KeyStore.SymmetricKey` *to* `Encrypted`

"Enforcing Security and Safety Models with an Information Flow Analysis Tool"

Assessment

- Existing Praxis tools check these contracts (recent KSU tools also infer them)
- While valuable, they are often too imprecise to describe realistic policies
- to verify more complex information flow properties, Rockwell Collins engineers previously manually constructed more precise verification models in the ACL2 theorem prover

Our work on conditional information flow thus has the potential to

- extend the expressiveness of SPARK info flow contracts to allow more precise reasoning at the source code level
- significantly increase the automation of constructing and checking information flow contracts

Overcoming SPARK Limitations

Original SPARK

```
--# derives w
--#    from y, z, x;
…
if x > 0 then
  w := y;
else
  w := z;
end if;
```

Enhanced SPARK (FM 08)

```
--# derives w
--#    from y when x > 0,
--#    from z when x <= 0,
--#    z;
```

Conditions on the pre-state allow us to more precisely describe flows

Many policies are conditional -- information is allowed to pass or is downgraded only in certain conditions

Analyzing Arrays

- Since SPARK has no heap, all complex data structures are coded as arrays.
- Yet arrays were analyzed as atomic entities (all flows are merged):
  - an update to $A[q]$ is treated as an update to A (all elements of A)
  - no way to say that, e.g., information at odd indices only flows to other odd index positions
- We want to reason about individual array elements.
- for assignment $A[Q] := E$, as in standard Hoare Logic [Gries], the precondition is found by substituting $A\{Q : E\}$ for A in postcondition.
- One can then simplify (and strengthen) the resulting precondition:

  Pre: $x = y \Rightarrow w \ltimes,\ x \neq y \Rightarrow A[y] \ltimes,\ (x = y) \ltimes$

  $$A[x] := w$$

  Post: $A[y] \ltimes$

Analyzing Loops

Always possible to make crude approximation:

1. consider arrays to be atomic entities
2. Iterate over assertions $\phi_x \Rightarrow x\ltimes$, weakening the antecedents
3. Use widening to ensure convergence (worst case: each ϕ_x becomes *true*)

But for certain **for** loops we can do better:

- many applications have loops that process elements independently of each other
- we can handle such loop in uniform way, by processing once with special symbolic variables that range over index values of variables, and then generalize (universally quantify)
- exists checks to detect loop-carried dependencies, but such tests can actually be expressed within our logic, by examining preconditions

For Loops, Simple Examples


```
--#derives
--#   forall u in {1..n}:
--#   A[u] from A[u+1]
--#and
--#   forall u notin {1..n}:
--#   A[u] from A[u]
for q ← 1 to n loop
    A[q] := A[q+1]
end loop
```

- not parallelizable
- but no loop-carried dependency
- precise analysis

```
--#derives A from *
for q ← 1 to n loop
    A[q] := A[q-1]
end loop
```

- not parallelizable
- and loop-carried dependency
- crude analysis

Analyzing For Loops (w/o Loop-Carried Deps)

for $q \leftarrow 1$ **to** m

$\quad t := A[q];\; A[q] := A[q+m];\; A[q+m] := t$

Find preconditions Θ for loop body B:

$$\{A[q+m]\ltimes\}\ B\ \{A[q]\ltimes\},\ \{A[q]\ltimes\}\ B\ \{A[q+m]\ltimes\}$$

We can now generate preconditions for $A[u]\ltimes$

$$\begin{aligned} u \in \{1..m\} &\Rightarrow A[u+m]\ltimes \\ u \in \{m+1..2m\} &\Rightarrow A[u-m]\ltimes \\ u \notin \{1..2m\} &\Rightarrow A[u]\ltimes \end{aligned}$$

Requirements that must be fulfilled:

1. q and $q+m$ not modified by loop body
2. No loop carried dependendies
   - on scalars: nothing in Θ modified except A
   - on array locations: cannot be read after being updated (this can be expressed precisely)
3. "inverses" (relating u and $q+m$) do exist

Correctness Proof

Automatically verified in Coq by Joey Dodds

- for the basic constructs: assignments, assertions, conditionals
- almost for **while** loops
- not yet for **for** loops

First approach:

- write a precondition analysis that generates witnesses
- prove in Coq that if a witness type checks with type $\{\Theta\}\ C\ \{\Theta'\}$ then this is indeed a semantically correct Hoare triple

Second approach: write the precondition generater inside Coq, and prove that it always generates correct evidence.

Analyzing Procedure Calls

Assume procedure p has contract

derives $A[u]$
from z **when** $u = x$
from $B[u]$ **when** $u \neq x$
from x
and w **from** z

Analyzing Procedure Calls

Assume procedure p has contract

derives $A[u]$
  from z **when** $u = x$
  from $B[u]$ **when** $u \neq x$
  from x
and w **from** z

$$y > 0 \land 7 = x \Rightarrow z\ltimes$$
$$y > 0 \land 7 \neq x \Rightarrow B[7]\ltimes$$
$$y > 0 \Rightarrow x\ltimes$$

$$\Big\uparrow \quad \textbf{call } p$$

$$y > 0 \Rightarrow A[7]\ltimes$$

Analyzing Procedure Calls

Assume procedure p has contract

derives $A[u]$
  from z **when** $u = x$
  from $B[u]$ **when** $u \neq x$
  from x
and w **from** z

$$y > 0 \wedge 7 = x \Rightarrow z\ltimes$$
$$y > 0 \wedge 7 \neq x \Rightarrow B[7]\ltimes$$
$$y > 0 \Rightarrow x\ltimes$$

$$true \Rightarrow y\ltimes$$

$\uparrow$ **call** p $\uparrow$

$$y > 0 \Rightarrow A[7]\ltimes \qquad w > 8 \Rightarrow y\ltimes$$

Analyzing Procedure Calls

Assume procedure p has contract

derives $A[u]$
  from z **when** $u = x$
  from $B[u]$ **when** $u \neq x$
  from x
and w **from** z

$$\begin{array}{ccc}
\begin{array}{l} y > 0 \wedge 7 = x \Rightarrow z\ltimes \\ y > 0 \wedge 7 \neq x \Rightarrow B[7]\ltimes \\ y > 0 \Rightarrow x\ltimes \end{array} & & z > 7 \Rightarrow y\ltimes \\
\uparrow & \textbf{call } p & \uparrow \text{ if } w = z_{\text{old}} + 1 \\
y > 0 \Rightarrow A[7]\ltimes & & w > 8 \Rightarrow y\ltimes
\end{array}$$

Analyzing Procedure Calls

Assume procedure p has contract

derives $A[u]$
  from z **when** $u = x$
  from $B[u]$ **when** $u \neq x$
  from x
and w **from** z

$y > 0 \wedge 7 = x \Rightarrow z\ltimes$
$y > 0 \wedge 7 \neq x \Rightarrow B[7]\ltimes$
$y > 0 \Rightarrow x\ltimes$

$z > 7 \Rightarrow y\ltimes$

call p

if $w = z_{\mathrm{old}} + 1$

$y > 0 \Rightarrow A[7]\ltimes$

$w > 8 \Rightarrow y\ltimes$

In the absence of functional contracts, experiments show significant precision loss.

Related Work

- Conditional declassification [Banerjee & Naumann & Rosenberg]
- Path conditions in program dependence graphs [Hammer, Krinke, Snelting etc]
- Type systems for information flow
- Work on SPARK information flow [Bergeretti & Carre; Chapman & Hilton]
- Information flow verification by self-composition

Comparison to Self-Composition

$$\{x \ltimes\}\ y := x + 2;\ \ w := y + 3\ \{w \ltimes\}$$

is equivalent to (using primes for fresh copies)

$$
\begin{array}{l}
\{x = x'\} \\
\qquad y := x + 2;\ \ w := y + 3 \\
\qquad y' := x' + 2;\ \ w' := y' + 3 \\
\{w = w'\}
\end{array}
$$

which may be checked by tool for standard safety analysis.

- must find intermediate assertions like $\{w = x' + 5\}$
- in general, need to find f such that $\{w = f(x')\}$
- for more complex dependencies, that may not be feasible unless the safety analysis "knows" that the program is generated by self-composition
- For good results, one therefore must combine with security static analysis [Terauchi/Aiken, SAS'05]