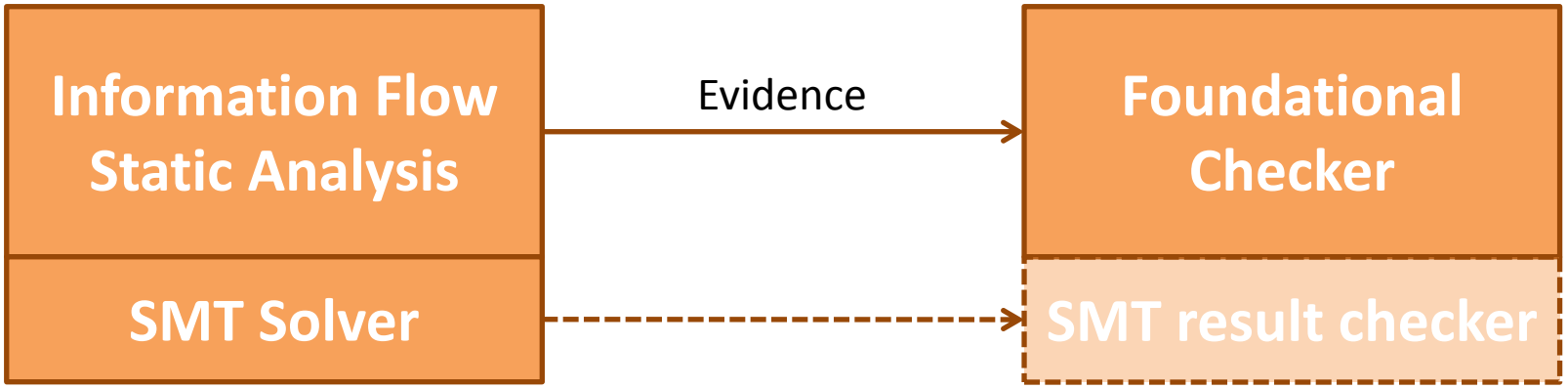
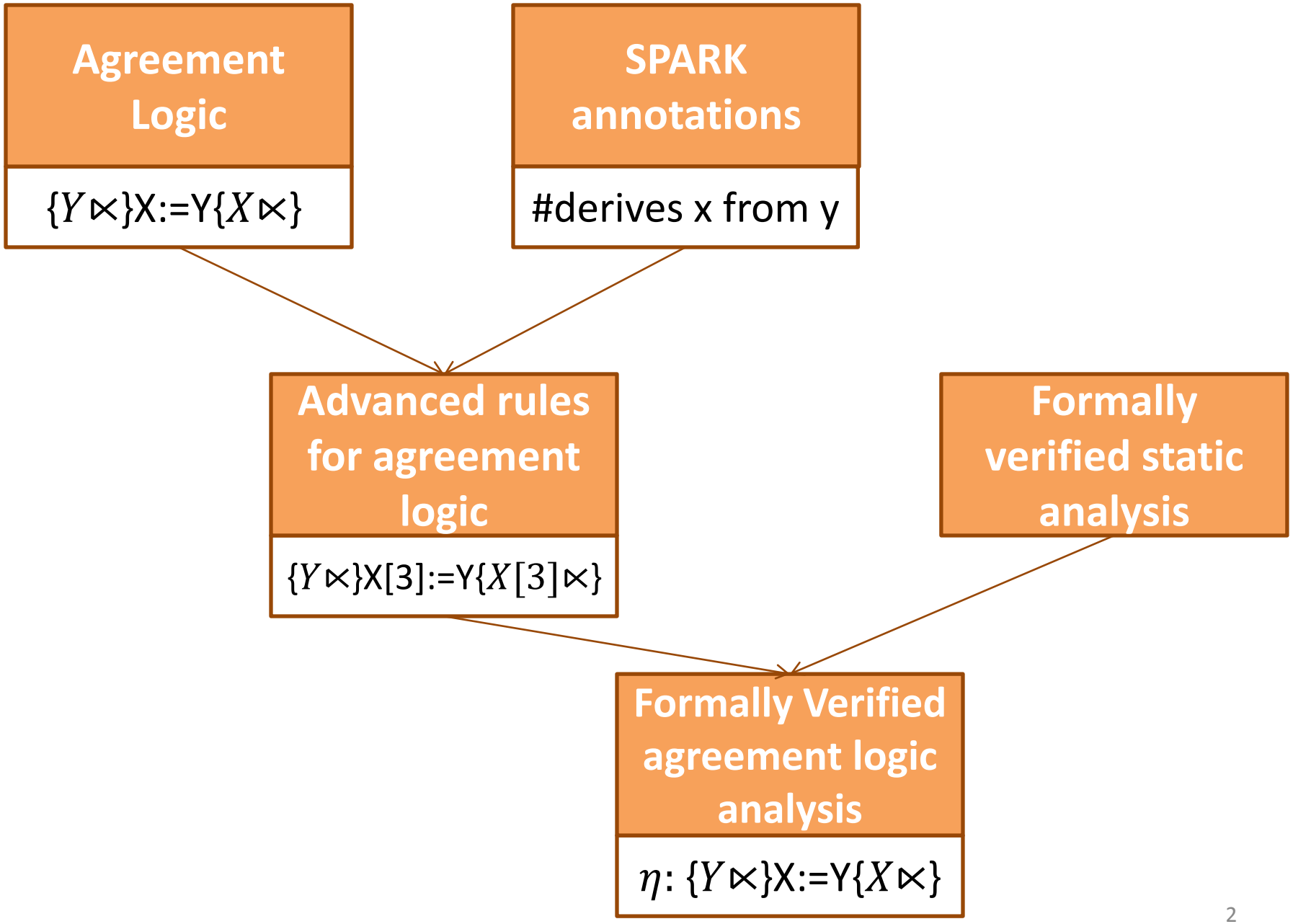
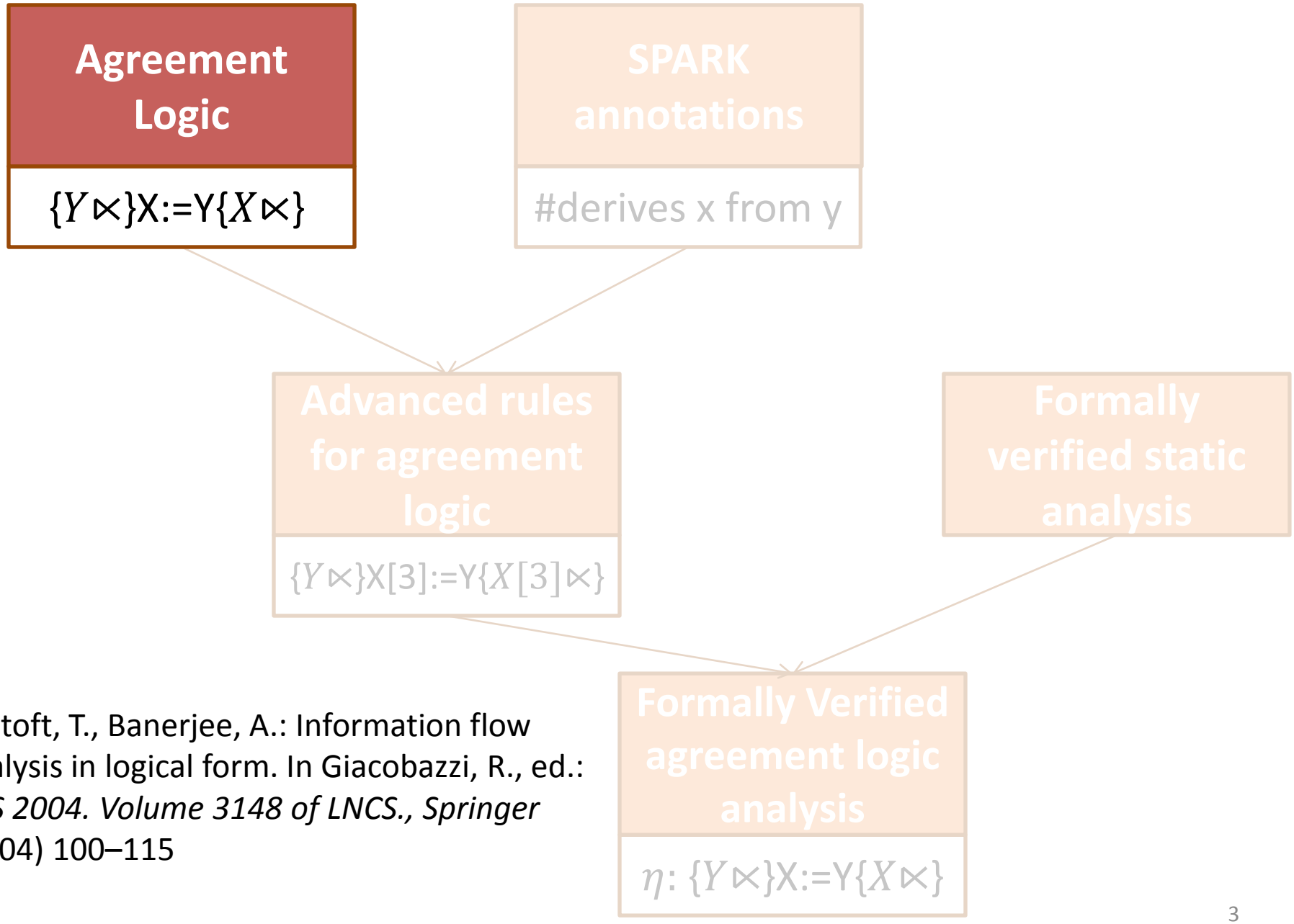
A Certificate Infrastructure for Machine-Checked Proofs of Conditional Information Flow

Torben Amtoft[K], Josiah Dodds[P], Zhi Zhang[K], Andrew Appel[P], Lennart Beringer[P], John Hatcliff[K], Xinming Ou[K], Andrew Cousino[K]

K = Kansas State University P = Princeton University




Agreement Logic
{Y⋉}X:=Y{X⋉}
SPARK annotations
#derives x from y
Advanced rules for agreement logic
{Y⋉}X[3]:=Y{X[3]⋉}
Formally verified static analysis
Formally Verified agreement logic analysis
η: {Y⋉}X:=Y{X⋉}

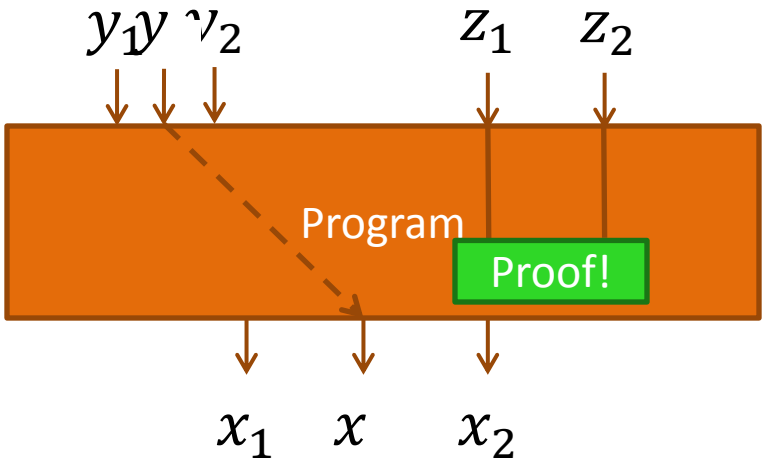


Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In Giacobazzi, R., ed.: *SAS 2004. Volume 3148 of LNCS., Springer* (2004) 100–115

Agreement Logic

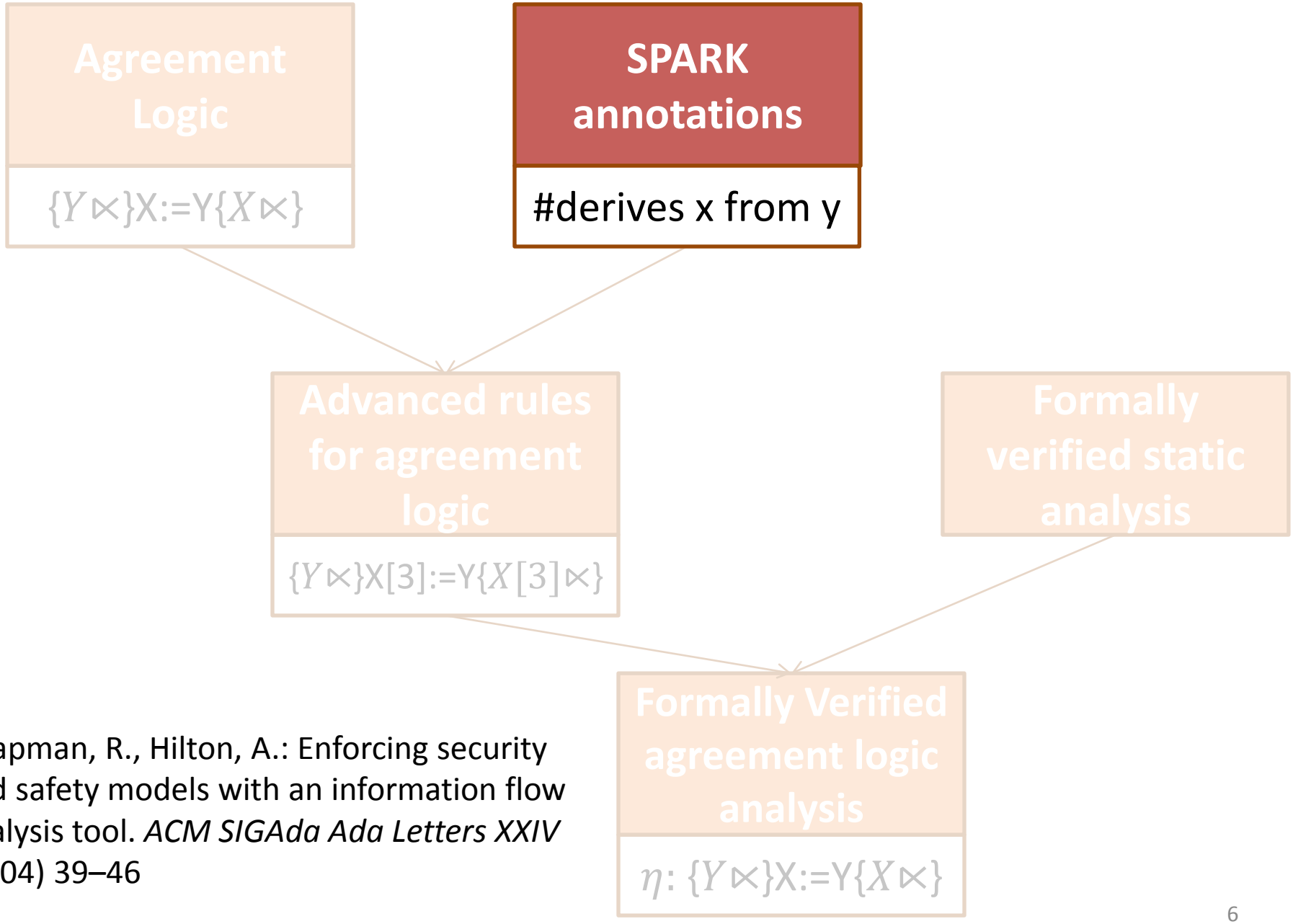
$S_1, S_2 \vDash x\ltimes$ means x has the same value when evaluated in S_1 and S_2

{y⋉} Program {x⋉}



$S_1, S_2 \vDash x \propto$

#derives x from y



Chapman, R., Hilton, A.: Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters XXIV* (2004) 39–46

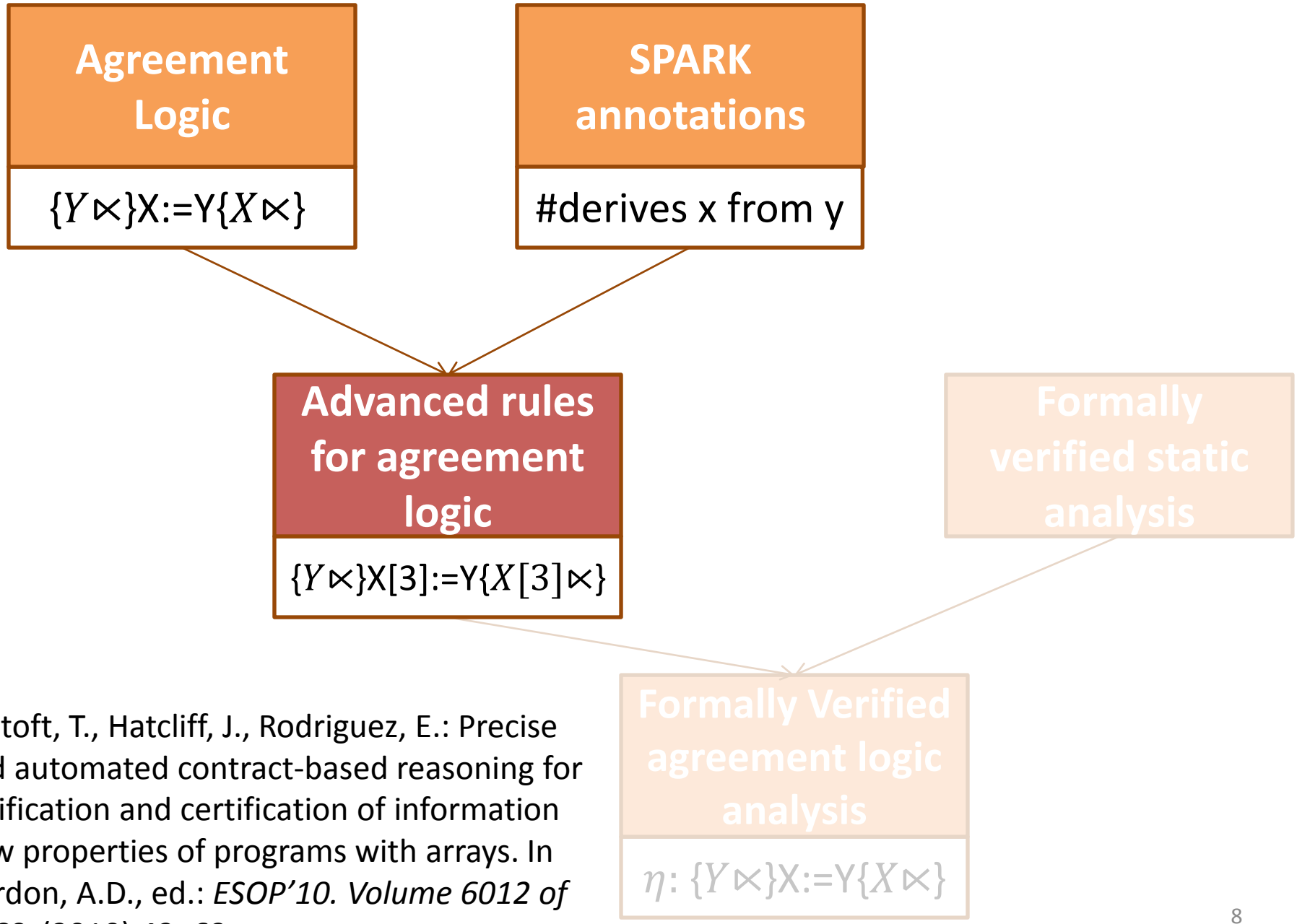
SPARK annotations

The SPARK programming language has annotations for information flow

The annotation `#derives x from y` means that only information from y might flow into x

SPARK has a static analysis to check this type of contract

These annotations are not always as expressive as programmers want



Amtoft, T., Hatcliff, J., Rodriguez, E.: Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. In Gordon, A.D., ed.: *ESOP'10. Volume 6012 of LNCS.* (2010) 43–63

Hoare-Style rules for agreement logic

inspired by SPARK annotations

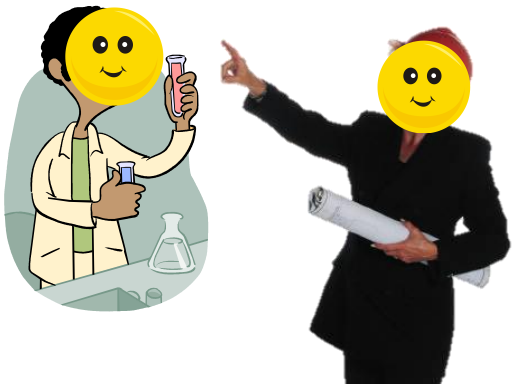
$\{y\ltimes\}\ x := y\ \{x\ltimes\}$

verifies

`#derives x from y`

on the program

$x := y$



Advanced rules for conditional information flow

More expressive rules

$\{b \Rightarrow y \ltimes \ \wedge \ \neg b \Rightarrow z \ltimes \ \wedge \ b \ltimes\}$
if b **then**
$x := y$
else
$x := z$
$\{x \ltimes\}$

Rules for reasoning about specific array locations, rather than entire arrays at once

Without conditional rules

More expressive rules

$\{b \Rightarrow y \ltimes \ \wedge \ \neg b \Rightarrow z \ltimes \ \wedge \ b \ltimes\}$
if b **then**
$x := y$
else
$x := z$
$\{x \ltimes\}$

Rules for reasoning about specific array locations, rather than entire arrays at once

D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. *TPHOLs 2004, volume 3223 of LNCS,* pages 34–49. Springer, 2004.

Y. Bertot. A Coq formalization of a type checker for object initialization in the Java virtual machine. *CAV'01*.

Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *POPL'06*. (2006) 42–54

Leroy, X. 2003. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning 30*, 235–269.

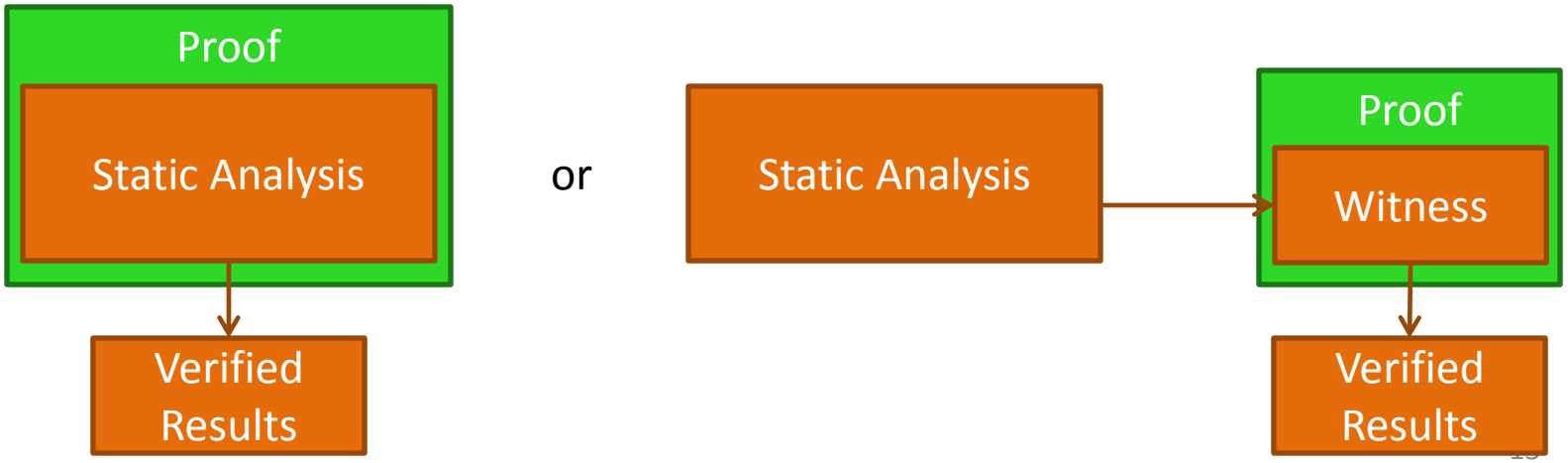
A.W. Appel. VeriSmall: Verified Smallfoot shape analysis. *CPP 2011: First International Conference on Certified Programs and Proofs*, (2011).

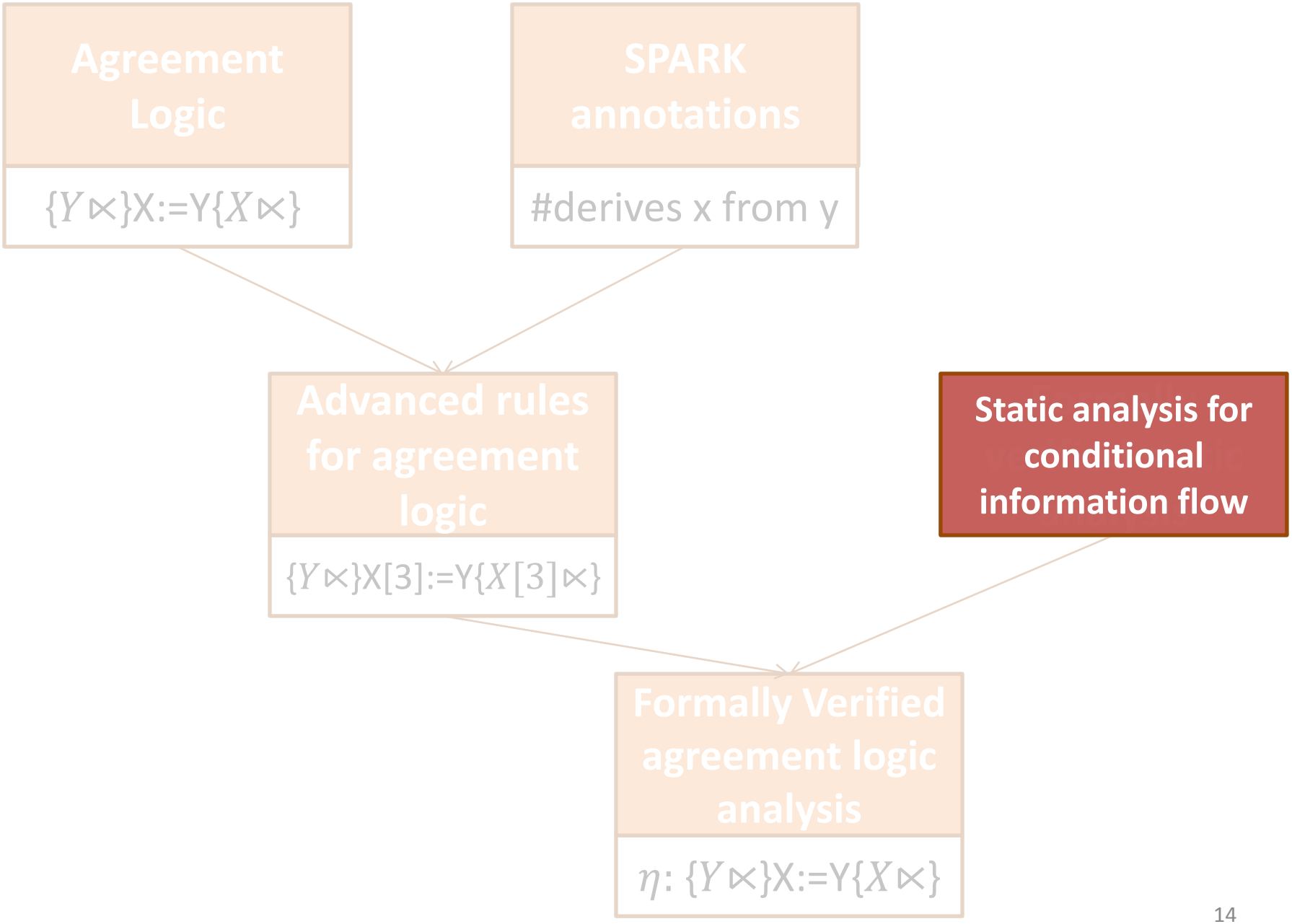
Formally verified static analysis

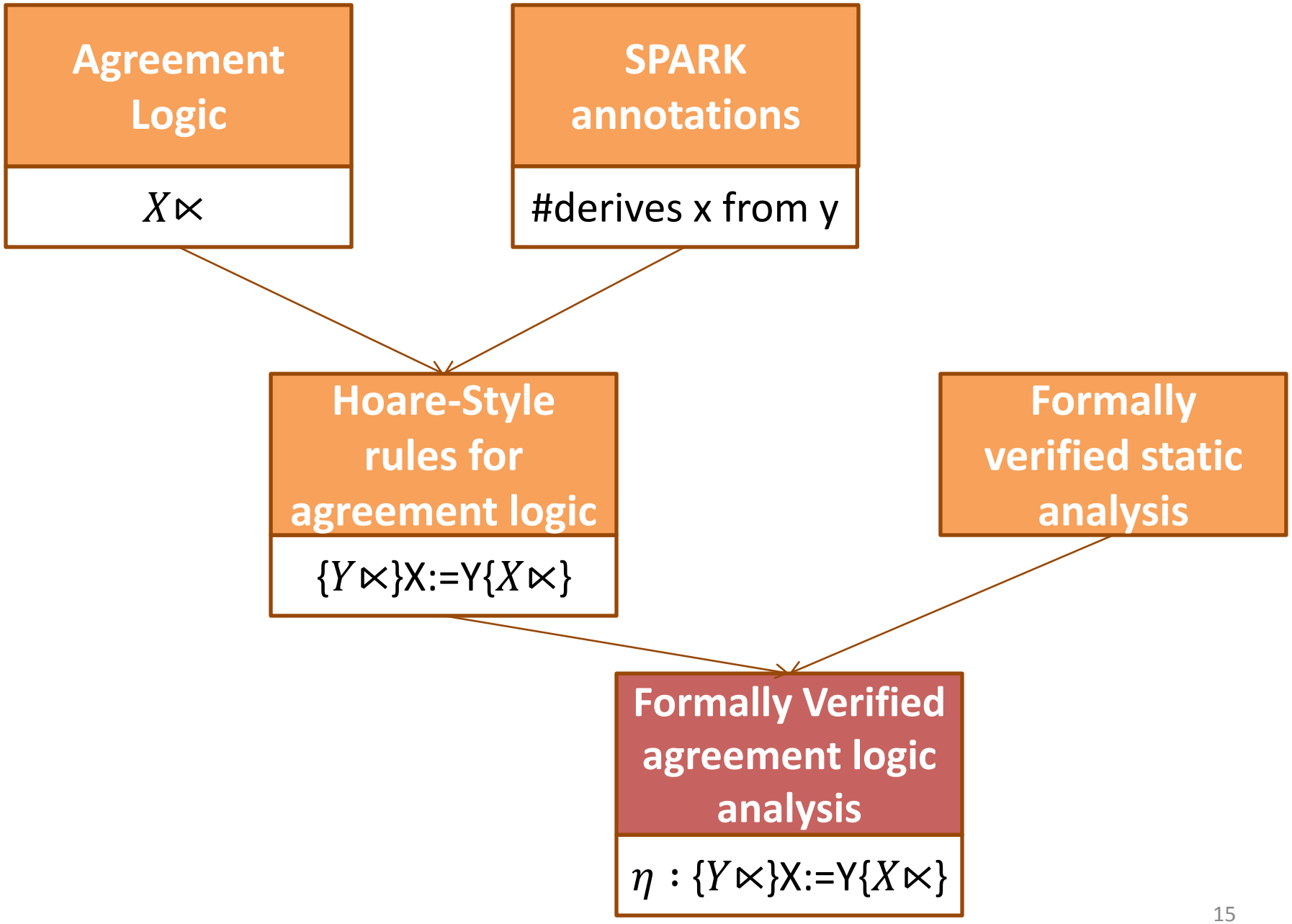
Many single-execution analyses

Formalizing the verifications require formalizing type-systems, operational semantics, and source languages

Soundness relates analyzer to operational semantics




Agreement Logic
{Y⋉}X:=Y{X⋉}
SPARK annotations
#derives x from y
Advanced rules for agreement logic
{Y⋉}X[3]:=Y{X[3]⋉}
Static analysis for conditional information flow
Formally Verified agreement logic analysis
η: {Y⋉}X:=Y{X⋉}

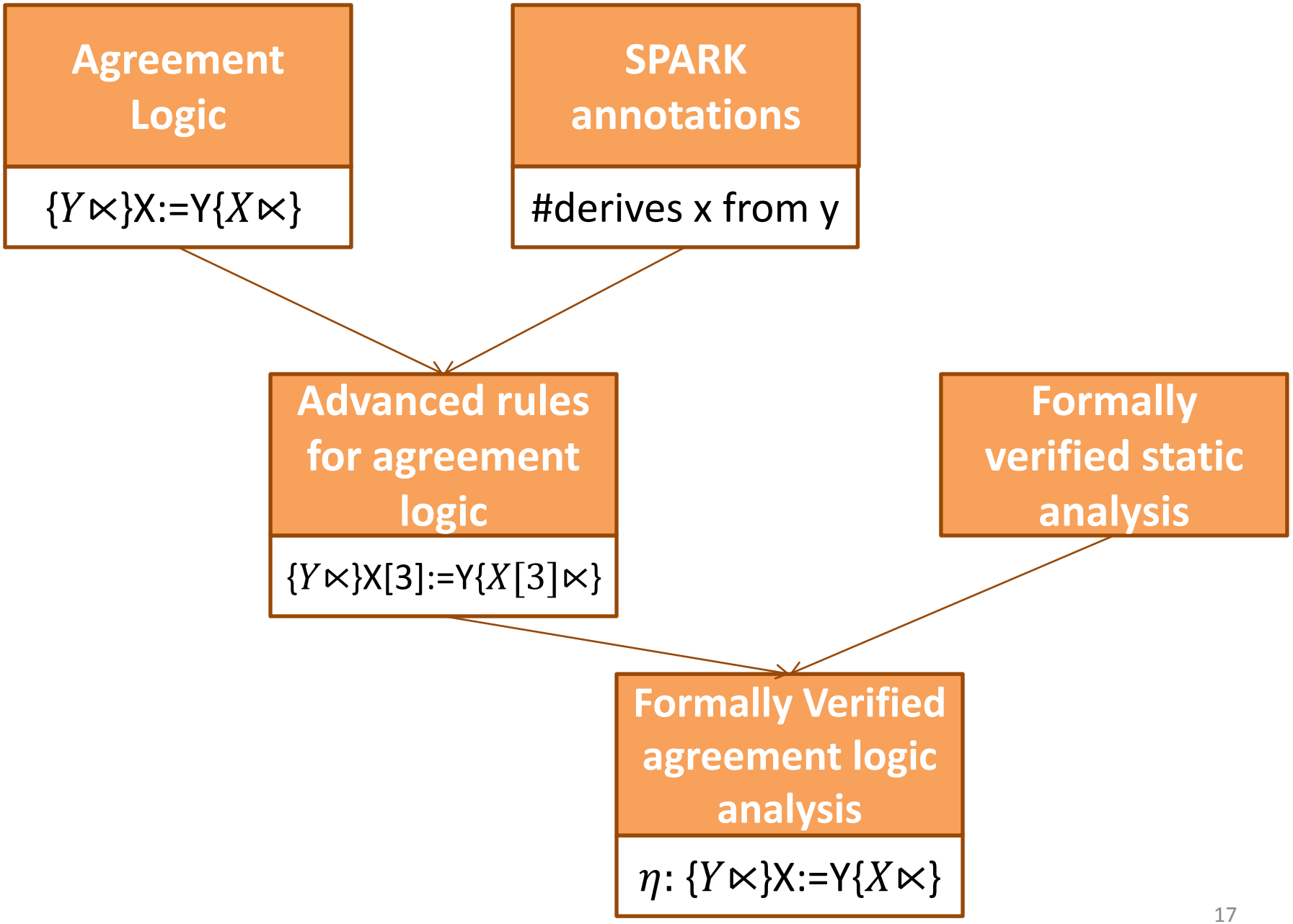

Agreement Logic
$X \ltimes$
SPARK annotations
#derives x from y
Hoare-Style rules for agreement logic
$\{Y \ltimes\}$X:=Y$\{X \ltimes\}$
Formally verified static analysis
Formally Verified agreement logic analysis
$\eta : \{Y \ltimes\}$X:=Y$\{X \ltimes\}$

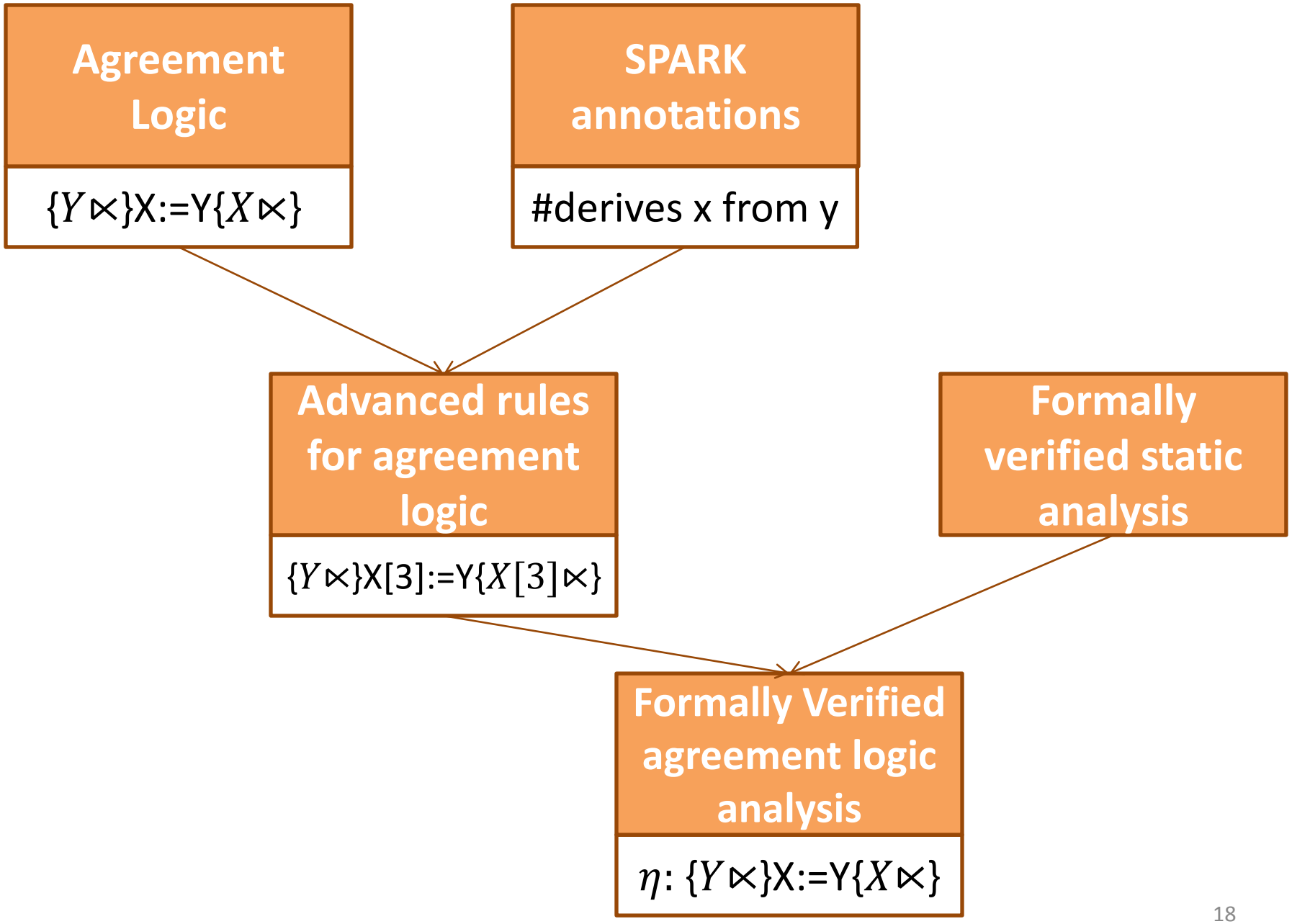
Contributions

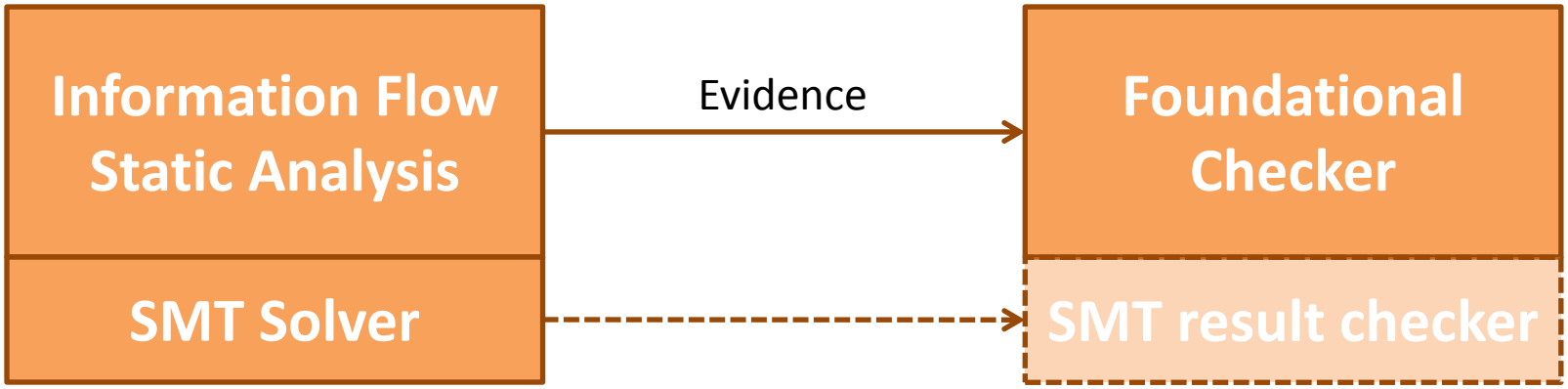
Implemented a static analysis with advanced rules for conditional information flow

Analysis generates witnesses to be formally verified in Coq

If the witness verifies, the result of the analysis is correct with respect to the operational semantics

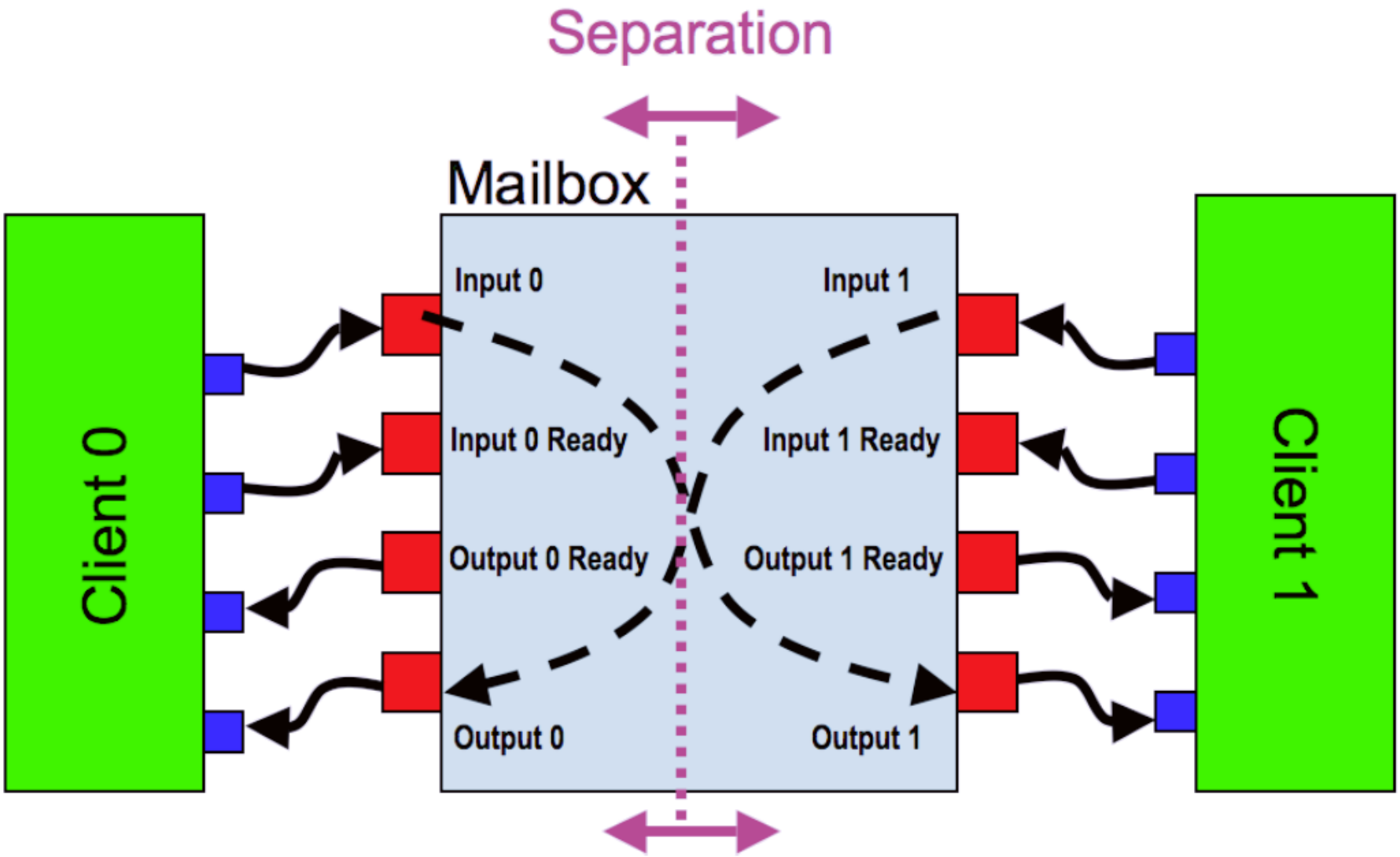

Agreement Logic
{Y∝}X:=Y{X∝}
SPARK annotations
#derives x from y
Advanced rules for agreement logic
{Y∝}X[3]:=Y{X[3]∝}
Formally verified static analysis
Formally Verified agreement logic analysis
η: {Y∝}X:=Y{X∝}


Agreement Logic
{Y⋉}X:=Y{X⋉}
SPARK annotations
#derives x from y
Advanced rules for agreement logic
{Y⋉}X[3]:=Y{X[3]⋉}
Formally verified static analysis
Formally Verified agreement logic analysis
η: {Y⋉}X:=Y{X⋉}



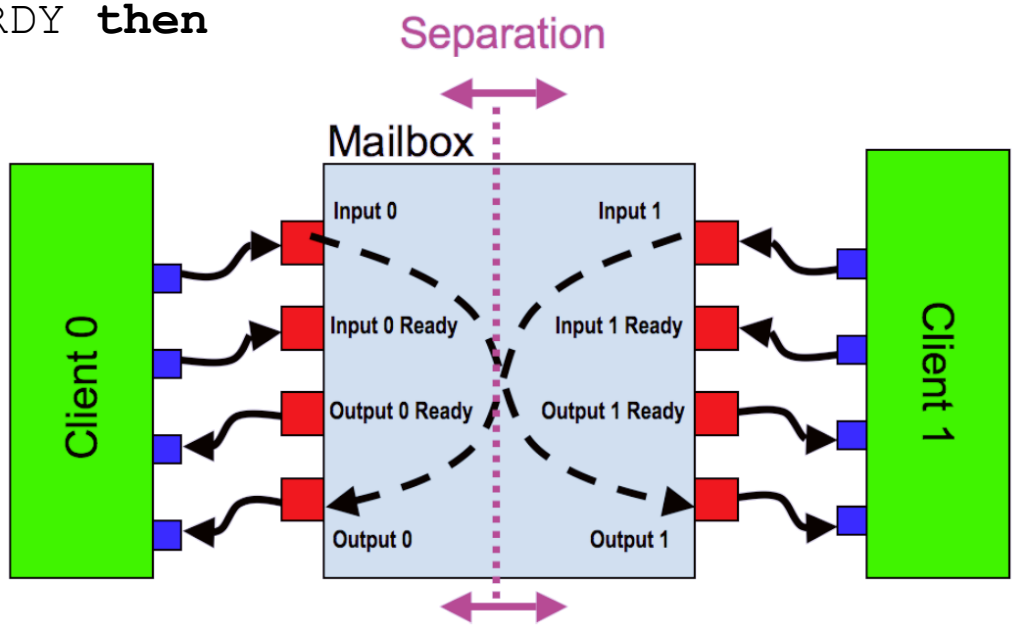
Michael Armand, Germain Faure, Benjamin Gregoire, Chantal Keller, Laurent Thíery & Benjamin Werner (2011): Verifying SAT and SMT in Coq for a fully automated decision procedure. In: PSATTT ’11

Example Application Program



Mailbox Example

```
if IN_1_RDY and not OUT_0_RDY then

        DATA_1 := IN_1 DAT;

        IN_1_RDY := false;

        OUT_0_DAT := DATA_1;

        OUT_0_RDY := true;

end if;
```



Mailbox Example

```
{IN_1_RDY ∧ OUT_0_RDY ⇒ IN_1_DAT⋉ ∧
 ¬IN_1_RDY ∨ OUT_0_RDY ⇒ OUT_0_DAT⋉ ∧
 IN_1_RDY⋉ ∧ OUT_0_RDY⋉}
if IN_1_RDY and not OUT_0_RDY then
       {IN_1_DAT⋉}
       DATA_1 := IN_1 DAT;
       {DATA_1⋉}
       IN_1_RDY := false;
       {DATA_1⋉}
       OUT_0_DAT := DATA_1;
       {OUT_0_DAT⋉}
       OUT_0_RDY := true;
       {OUT_0_DAT⋉}
end if;
{OUT_0_DAT⋉}
```

2-Assertions

$\mathrm{E}\ltimes$

2-Assertions

$$S_1, S_2 \vDash \mathrm{E}\ltimes$$

2-Assertions

$$\frac{[\![E]\!]_{S_1} = [\![E]\!]_{S_2}}{S_1, S_2 \vDash \mathrm{E}\ltimes}$$

2-Assertions

$$S_1, S_2 \vDash \phi \Rightarrow \mathrm{E}\ltimes$$

2-Assertions

$$\frac{(\neg[\![\phi]\!]_{S_1} \vee \neg[\![\phi]\!]_{S_2}) \quad \vee \quad [\![E]\!]_{S_1} = [\![E]\!]_{S_2}}{S_1, S_2 \vDash \phi \Rightarrow \mathrm{E}\propto}$$

Triples

$$\{\Theta\}\ C\ \{\Theta'\}$$

Holds when

$$\forall\ S_1\ S_2\ S_1'\ S_2'.$$

$$S_1, S_2 \vDash \Theta \quad \Rightarrow \quad S_1[\![C]\!]\ S_2 \quad \Rightarrow \quad S_1'[\![C]\!]S_2'$$

$$\Rightarrow \quad S_1', S_2' \vDash \Theta'$$

Hoare Triple

```
Fixpoint validHoareTriple
(X: list SkalVar) C (asns: assns):=
  match asns with
  | Assns Θ Θ′ =>
    (forall S1 S2 S1′ S2′, lookupsComplete X S1 S2
     ->
        (twoAssnsInterpretation S1 S2 Θ) ->
       Opsem S1 C S2 -> Opsem S1′ C S2′ ->
       (twoAssnsInterpretation S1′ S2′ Θ′))
  |  …
```

$$\forall\, S_1\, S_2\, S_1'\, S_2'.$$

$$S_1, S_2 \vDash \Theta \quad \Rightarrow \quad S_1 [\![C]\!]\, S_2 \quad \Rightarrow \quad S_1' [\![C]\!] S_2'$$

$$\Rightarrow \quad S_1', S_2' \vDash \Theta'$$

(twoAssnsInterpretation $S_1\, S_2\, \Theta$) ->
Opsem S_1 C S_2 -> Opsem S_1' C S_2' ->
(twoAssnsInterpretation $S_1'\, S_2'\, \Theta'$))

Evidence

Evidence takes the form

$$\vdash \eta : \{\Theta\}\ C\ \{\Theta'\}$$

If η is well typed and the evidence statements are sound wrt. the operational semantics of our language we have a valid triple. Each form of evidence corresponds directly to a coq constructor.

$$\{\Theta\}\ C\ \{\Theta'\}$$

Rules

$$\frac{}{\vdash \mathrm{AssignE}(\Theta, x, A) : \{\Theta[A/x]\}\ x := A\{\Theta\}}$$

$$\frac{}{\vdash \mathrm{SkipE}(\Theta) : \{\Theta\}\ \mathbf{Skip}\ \{\Theta\}}$$

$$\frac{}{\vdash \mathrm{AssertE}(B) : \{B \Rightarrow \Theta\}\ \mathbf{Assert}(B)\{\Theta\}}$$

Conditional rule

$$\frac{\vdash \eta_1 : \{\Theta_1\}\ C_1\ \{\phi' \Rightarrow E \ltimes\} \quad \vdash \eta_2 : \{\Theta_2\}\ C_2\ \{\phi' \Rightarrow E \ltimes\} \quad \vdash \nu : \phi \overset{C}{\Leftarrow} \phi'}{\vdash \mathrm{CondE}(\eta_1, \eta_2, \nu,\ B) : \{B \Rightarrow \Theta_1 \wedge (\neg B \Rightarrow \Theta_2) \wedge (\phi \Rightarrow B \ltimes)\}\ C\ \{\phi' \Rightarrow E \ltimes\}}$$

$C = \boldsymbol{if}\ B\ \boldsymbol{then}\ C_1\ \boldsymbol{else}\ C_2$

Evidence exists for the triple
$\{B \Rightarrow \Theta_1 \quad \wedge \quad \neg B \Rightarrow \Theta_2 \quad \wedge \quad \phi \Rightarrow B \ltimes\}\ C\ \{\phi' \Rightarrow E \ltimes\}$
if

Evidence exists for both branches of C
We have Necessary Precondition (NPC) evidence for ϕ and ϕ' across C

Coq Evidence Type

Evidence is an inductive proposition:

```
Inductive TEvid (X: list SkalVar) :
   Command -> assns -> Prop :=
| TSkipE ...
| TAAssignE ...
| TCondE ...
...
```

Coq Evidence Type

```
| TAAssignE : forall Theta x A,
    TEvid X (Assign x (AExp A))
     (Assns(TwoAssnsSubstA Theta x A) Theta)
```

$$\frac{}{\vdash \mathrm{AssignE}(\Theta, x, A) : \{\Theta[A/x]\}\, x := A\{\Theta\}}$$

Evidence Soundness

$$\vdash \eta : \{\Theta\}\ C\ \{\Theta'\} \Rightarrow$$

$$(\forall\ S_1\ S_2\ S_1'\ S_2'.$$

$$S_1, S_2 \vDash \Theta \quad \Rightarrow \quad S_1 [\![C]\!]\ S_2 \quad \Rightarrow \quad S_1' [\![C]\!] S_2' \Rightarrow$$

$$S_1, S_2 \vDash \Theta')$$

Soundness Statement

```
Theorem soundness :
forall X {C asns}, TEvid X C asns ->
(validHoareTriple X C asns)
```

$$\forall\, S_1\, S_2\, S_1'\, S_2'.$$

$$S_1, S_2 \vDash \Theta \quad \Rightarrow \quad S_1 [\![C]\!]\, S_2 \quad \Rightarrow \quad S_1' [\![C]\!] S_2' \Rightarrow$$

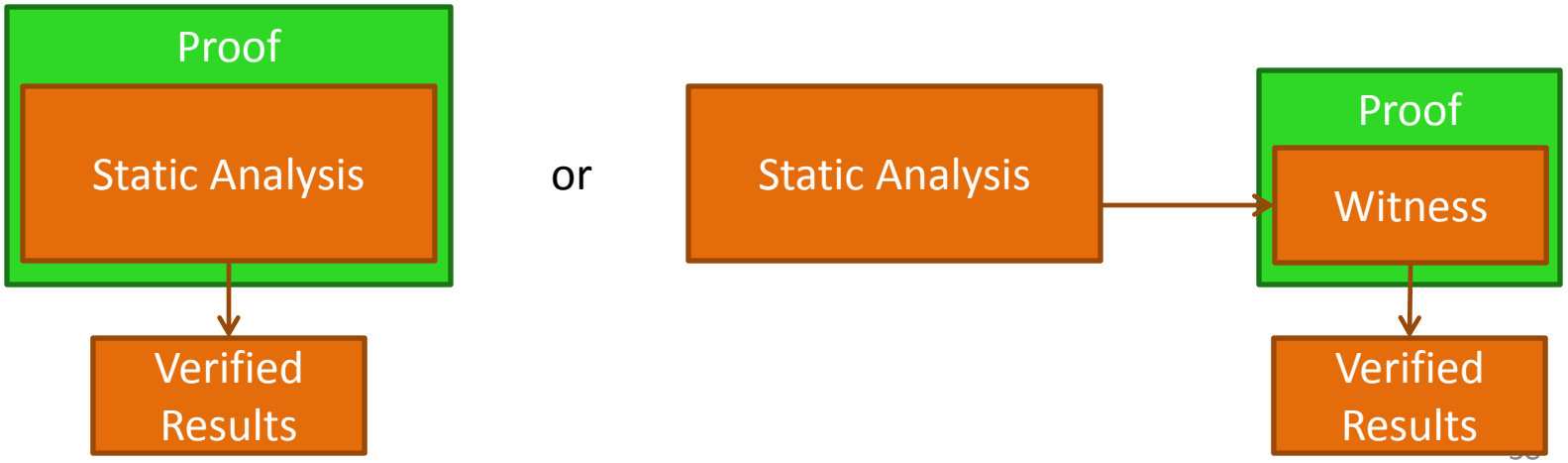
$$S_1, S_2 \vDash \Theta'$$

Paper proof is done. Coq proof is complete for all but the loop rules `WhileE` and `ForPolyE`

Precondition Generator in Coq

Evidence is large

Move from generating a witness to implementing algorithm



Future Work

Complete Coq proofs

Verify SMT results

Finish Coq precondition generator

Hook up to a certified compiler for an end-to-end system

Evaluate the tool on more real-world examples

Questions

Assignment Rule

$$\vdash \mathrm{AssignE}(T, x, A) : \{T[A/x]\}\ x := A\{T\}$$

Example

{y#}

x:=y

{x#}

Post-condition form

The post-condition of some rules takes the form

$$\phi \Rightarrow E \ltimes$$

Post-conditions of this form specify the information-flow properties we are interested in

Post-condition form

Our algorithm generates triples of the form

$$\{ B_1 \Rightarrow E_1 \ltimes \wedge \ \ldots \ \wedge B_n \Rightarrow E_n \ltimes \}\ C\ \{x \ltimes\}$$

The values of $E_1 \ldots E_n$ might flow into variable x when the respective $B_i \ldots B_n$ hold

We do this for each x modified in C

Conditional

$$C = \textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ C_1\ \textbf{\textit{else}}\ C_2$$

$$\frac{\vdash \eta_1 : \{T_1\}\ C_1\ \{B'_2 \Rightarrow E \ltimes\} \quad \vdash \eta_2 : \{T_2\}\ C_2\ \{B'_2 \Rightarrow E \ltimes\} \quad \vdash \nu : B_2 \underset{\Leftarrow}{\overset{C}{}} B'_2}{\vdash \mathrm{CondE}(\eta_1, \eta_2, \nu, B) : \{B \Rightarrow T_1 \wedge (\neg B \Rightarrow T_2) \wedge (B_2 \Rightarrow B \ltimes)\}\ C\ \{B'_2 \Rightarrow E \ltimes\}}$$

Conditional Example

```
{secure#, secure => low#,
!secure => hi#}
if secure then
   {low}
   hi := low
else
   {hi#}
   low := hi
fi
{low#}
```

Checking the Evidence

We said earlier that $\vdash \eta : \{T\}\ C\ \{T'\}$ is a proof that we have generated a valid triple if η is typed correctly and if our evidence is proven sound.

To actually verify this we define evidence as a coq type and prove it sound with respect to the defined operational semantics

Coq Evidence

We use a deep embedding of the logic and programming language

The operational semantics of our core language are defined in coq

Two assertions ($S_1, S_2 \vdash B \Rightarrow E \ltimes$)

```
Inductive twoSatisfies(s1
s2:State): TwoAssn -> Prop :=
|TwoSatisfies: forall B E,
   BEval B s1 = Some true ->
   BEval B s2 = Some true ->
   (Eval E s1 = Eval E s2) ->
   twoSatisfies s1 s2 (B, E).
```

Precondition Generator in Coq

```
Fixpoint generatePrecondition
    (c:Command) (post:TwoAssns)
    (X: list SkalVar):
    (TwoAssns * list SkalVar) := ...

Theorem generatePreconditionEvidence:
   forall C Y X Z post pre,
   (pre, X)=generatePrecondition C
  post Z ->
   allVarsIn X Y = true->
   TEvid Y C (Assns pre post).
```

Conclusions

Fully automated and formally verified information flow contract checker

Piece of an eventual end-to-end system

…

End

Semantics

$$S \vDash B \Leftrightarrow [\![B]\!]_S = true$$

$$\frac{S_1 \vDash B \quad S_2 \vDash B}{S_1, S_2 \vDash B} \qquad \frac{[\![B]\!]_{S_1} = [\![B]\!]_{S_2}}{S_1, S_2 \vDash B\ltimes}$$

$$\frac{S_1, S_2 \vDash \neg T_1 \vee S_1, S_2, \vDash T_2}{S_1, S_2 \vDash T_1 \Rightarrow T_2}$$

Previous work

SIFL precondition algorithm to check conditional information flow policies

Algorithm uses an agreement logic with hoare-style rules

Allows for precise treatment of arrays and for loops