Language-based methods for software security

Gilles Barthe

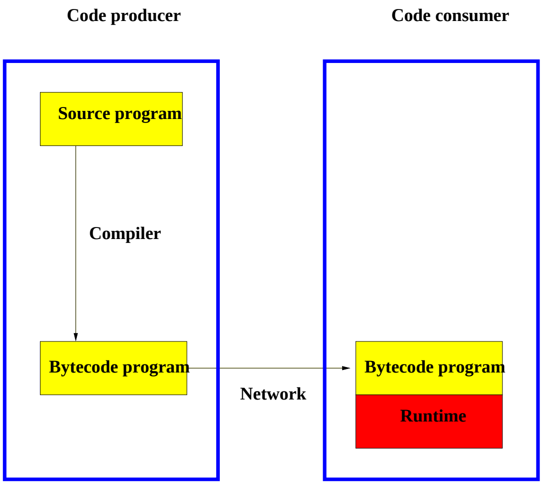
IMDEA Software, Madrid, Spain

`Part 1`

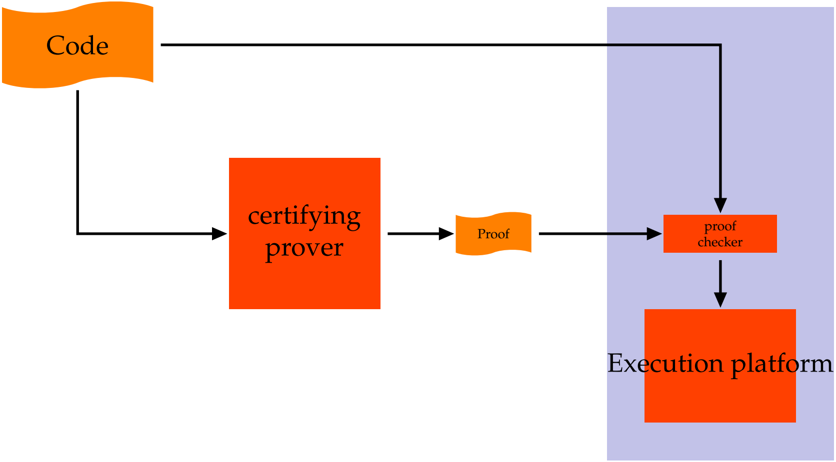
Motivation

- Mobile code is ubiquitous: large distributed networks of JVM devices
  - aimed at providing a global and uniform access to services
  - provide support to untrusted mobile code
- Security is a central concern: untrusted code may
  - use too many resources
    - CPU, memory...
  - perform unauthorized actions
    - open sockets
  - be hostile towards other applications
    - access, manipulate or reveal sensitive data
  - crash the system
    - destruction/corruption of files

Security challenge



Proof carrying code: principles

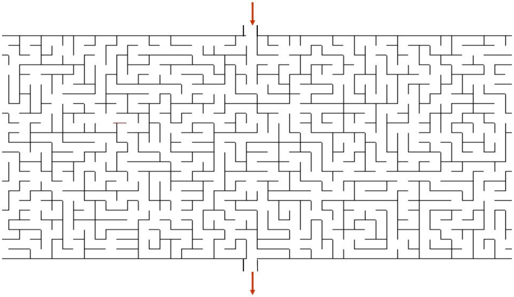


Certificates

- are condensed and formalized mathematical proofs/hints
- are self-evident and unforgeable
- can be checked efficiently. . .
- independent of difficulty of certificate generation

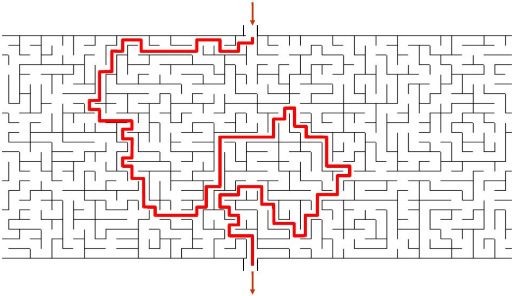
Certificates

- are condensed and formalized mathematical proofs/hints
- are self-evident and unforgeable
- can be checked efficiently. . .
- independent of difficulty of certificate generation



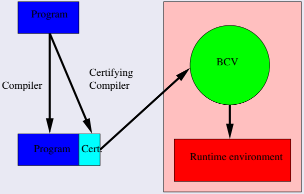
Certificates

- are condensed and formalized mathematical proofs/hints
- are self-evident and unforgeable
- can be checked efficiently...
- independent of difficulty of certificate generation



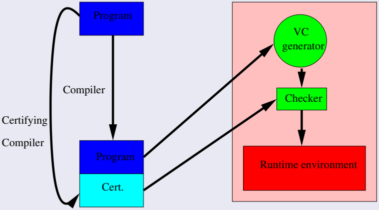
Flavors of Proof Carrying Code

Type-based PCC



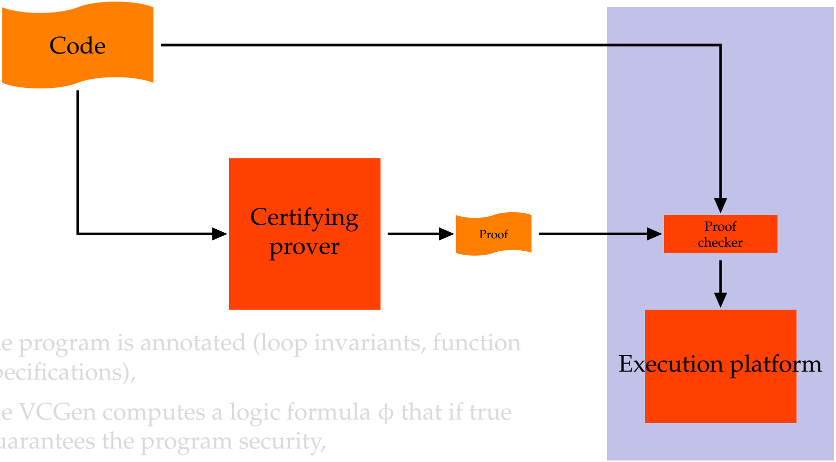
- Widely deployed in KVM
- Application to JVM typing
- On-device checking possible

Logic-based PCC



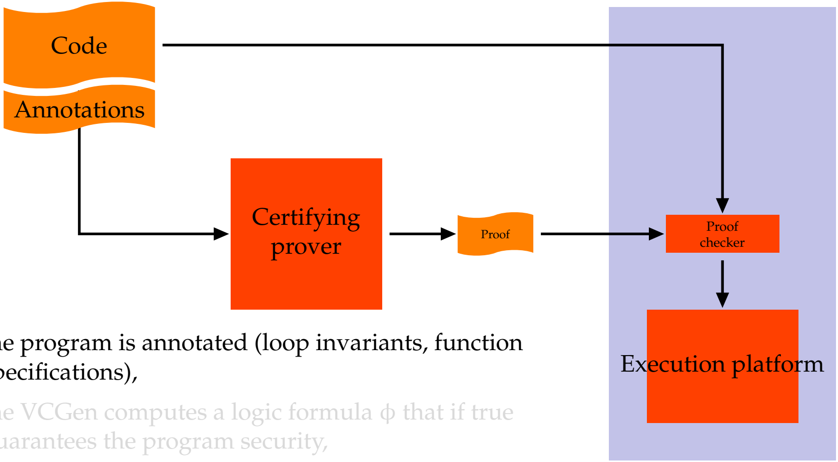
- Original scenario
- Application to type safety and memory safety

Proof carrying code: standard framework



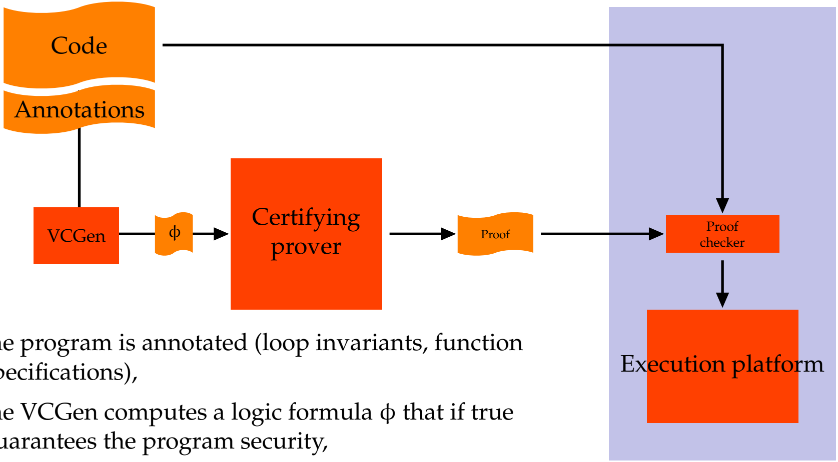
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π is a valid proof of ϕ.

Proof carrying code: standard framework



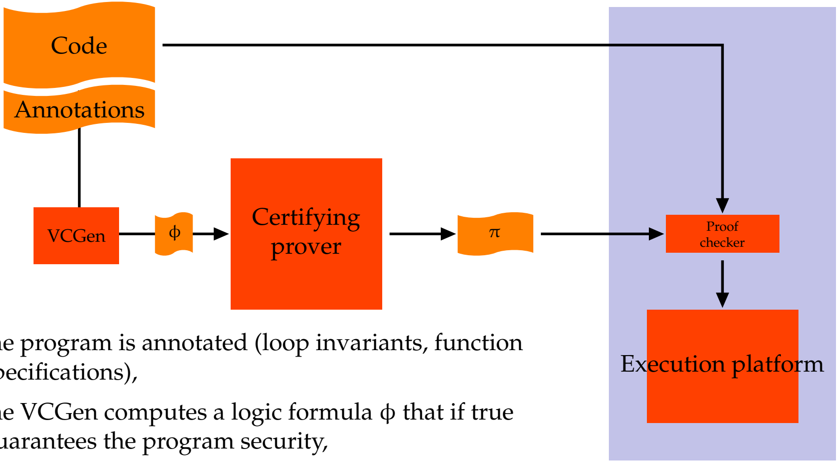
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π is a valid proof of ϕ.

Proof carrying code: standard framework



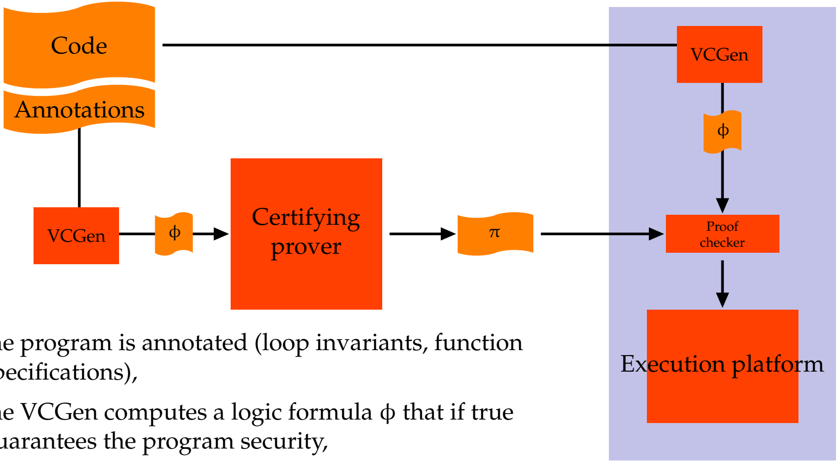
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π is a valid proof of ϕ.

Proof carrying code: standard framework



- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π is a valid proof of ϕ.

Proof carrying code: standard framework



- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π is a valid proof of ϕ.

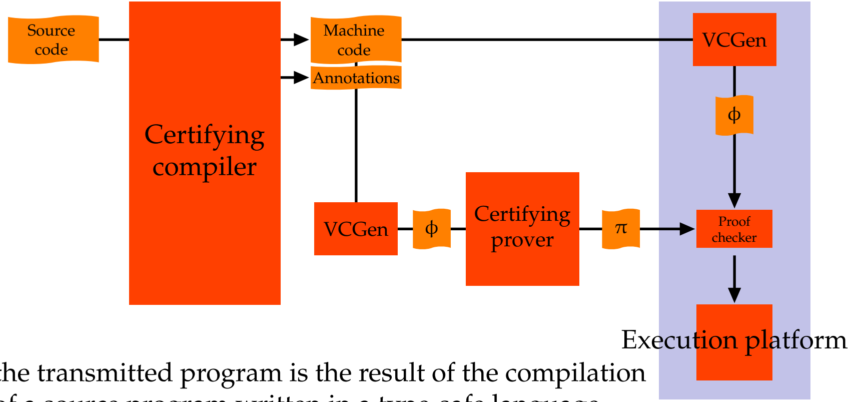
Certifying prover

- automatically proves the verification conditions (VC)
  - VC must fall in some logic fragments whose decision procedures have been implemented in the prover
- in the PCC context, proving is not sufficient, detailed proof must be generated too
  - like decision procedures in skeptical proof assistants
  - proof producing decision procedures are more and more considered as an important software engineering practice to develop proof assistants

Touchstone's certifying prover includes

- congruence closure and linear arithmetic decision procedures
- with a Nelson-Oppen architecture for cooperating decision procedures

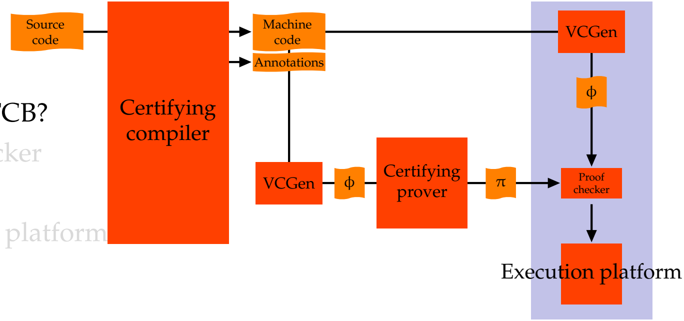
Annotation generation



- the transmitted program is the result of the compilation of a source program written in a type-safe language
- the role of the certifying compiler is
  - to check type-safety of the source program
  - to generate corresponding annotations in the machine code to help the VCGen

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.



What is the PCC TCB?

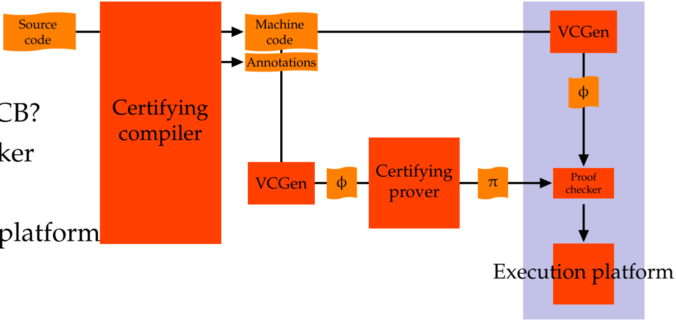
- the proof checker
- the VCGen
- the Execution platform

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?

- the proof checker
- the VCGen
- the Execution platform

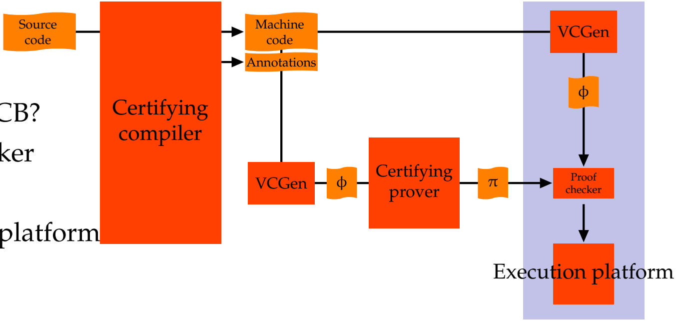


Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?

- the proof checker
- the VCGen
- the Execution platform



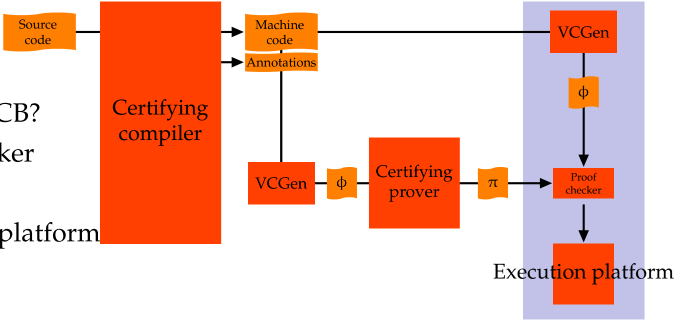
You don't need to trust ...

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?

- the proof checker
- the VCGen
- the Execution platform



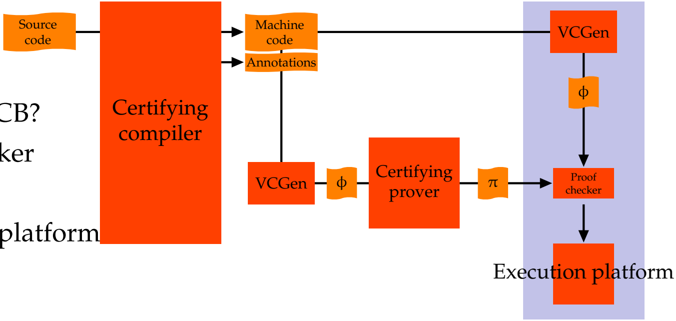
You don't need to trust the compiler ...

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?

- the proof checker
- the VCGen
- the Execution platform



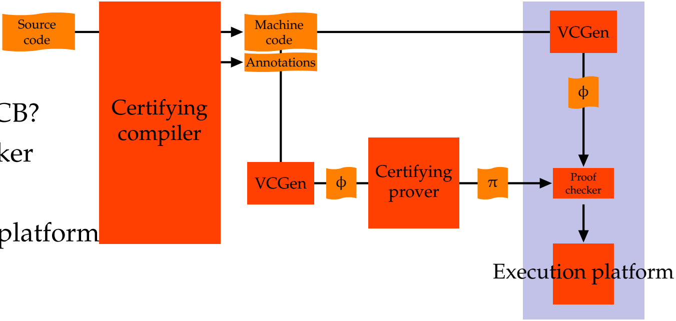
You don't need to trust the compiler, the annotations ...

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?

- the proof checker
- the VCGen
- the Execution platform



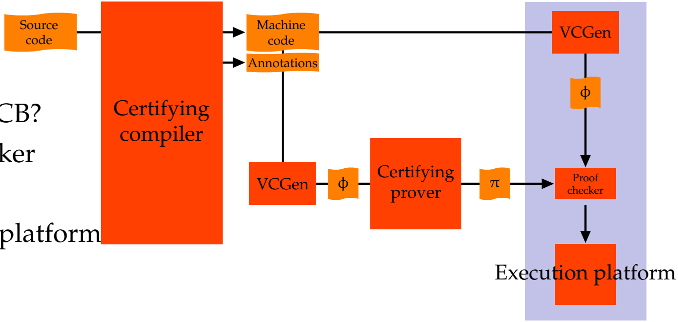
You don't need to trust the compiler, the annotations, the prover ...

Trusted Computing Base (TCB)

The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?

- the proof checker
- the VCGen
- the Execution platform



You don't need to trust the compiler, the annotations, the prover, the proof ...

Other instances of PCC

- Touchstone has achieved an impressive level of scalability (programs with about one million instructions)
- but[1] *"[...], there were errors in that code that escaped the thorough testing of the infrastructure"*.
- the weak point was the VCGen (23,000 lines of C...)

The size of the TCB can be reduced

1. by relying on simpler checkers
2. by removing the VCGen: *Foundational Proof-Carrying Code*
3. by certifying the VCGen in a proof assistant

[1] G.C. Necula and R.R. Schneck. *A Sound Framework for Untrusted Verification-Condition Generators.* LICS'03

Proof

$$\ddot{\alpha}[\![P]\!](\mathrm{Post}[\![\texttt{if}\ B\ \texttt{then}\ S_t\ \texttt{else}\ S_f\ \texttt{fi}]\!])$$

$$= \quad \wr\text{def. (110) of } \ddot{\alpha}[\![P]\!]\wr$$

$$\ddot{\alpha}[\![P]\!] \circ \mathrm{Post}[\![\texttt{if}\ B\ \texttt{then}\ S_t\ \texttt{else}\ S_f\ \texttt{fi}]\!] \circ \ddot{\gamma}[\![P]\!]$$

$$= \quad \wr\text{def. (103) of Post}\wr$$

$$\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[\tau^\star[\![\texttt{if}\ B\ \texttt{then}\ S_t\ \texttt{else}\ S_f\ \texttt{fi}]\!]] \circ \ddot{\gamma}[\![P]\!]$$

$$= \quad \wr\text{big step operational semantics (93)}\wr$$

$$\ddot{\alpha}[\![P]\!] \circ \mathrm{post}\,[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t) \cup (1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)] \circ \ddot{\gamma}[\![P]\!]$$

$$= \quad \wr\text{Galois connection (98) so that post preserves joins}\wr$$

$$\ddot{\alpha}[\![P]\!] \circ (\mathrm{post}\,[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t)] \mathbin{\dot{\cup}} \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)]) \circ \ddot{\gamma}[\![P]\!]$$

$$= \quad \wr\text{Galois connection (106) so that } \ddot{\alpha}[\![P]\!] \text{ preserves joins}\wr$$

$$(\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t)] \circ \ddot{\gamma}[\![P]\!]) \mathbin{\dot{\sqcup}} (\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)] \circ \ddot{\gamma}[\![P]\!])$$

$$\mathrel{\dot{\ddot{\sqsubseteq}}} \quad \wr\text{lemma (5.3) and similar one for the } \texttt{else} \text{ branch}\wr$$

$$\begin{aligned}
&\lambda J \bullet \mathrm{let}\ J^{t'} = \lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \mathrm{at}_P[\![S_t]\!]\ \textbf{?}\ J_{\mathrm{at}_P[\![S_t]\!]} \mathbin{\dot{\sqcup}} \mathrm{Abexp}[\![B]\!](J_\ell)\ \text{¿}\ J_l)\ \mathrm{in} \\
&\qquad \mathrm{let}\ J^{t''} = \mathrm{APost}[\![S_t]\!](J^{t'})\ \mathrm{in} \\
&\qquad\quad \lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \ell'\ \textbf{?}\ J^{t''}_{\ell'} \mathbin{\dot{\sqcup}} J^{t''}_{\mathrm{after}_P[\![S_t]\!]}\ \text{¿}\ J^{t''}_l) \\
&\quad \ddot{\sqcup} \\
&\quad \mathrm{let}\ J^{f'} = \lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \mathrm{at}_P[\![S_f]\!]\ \textbf{?}\ J_{\mathrm{at}_P[\![S_f]\!]} \mathbin{\dot{\sqcup}} \mathrm{Abexp}[\![T(\neg B)]\!](J_\ell)\ \text{¿}\ J_l)\ \mathrm{in} \\
&\qquad \mathrm{let}\ J^{f''} = \mathrm{APost}[\![S_f]\!](J^{f'})\ \mathrm{in} \\
&\qquad\quad \lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \ell'\ \textbf{?}\ J^{f''}_{\ell'} \mathbin{\dot{\sqcup}} J^{f''}_{\mathrm{after}_P[\![S_f]\!]}\ \text{¿}\ J^{f''}_l)
\end{aligned} \tag{120}$$

$$= \quad \wr\text{by grouping similar terms}\wr$$

$$\begin{aligned}
&\lambda J \bullet \mathrm{let}\ \ J^{t'} = \lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \mathrm{at}_P[\![S_t]\!]\ \textbf{?}\ J_{\mathrm{at}_P[\![S_t]\!]} \mathbin{\dot{\sqcup}} \mathrm{Abexp}[\![B]\!](J_\ell)\ \text{¿}\ J_l) \\
&\quad \mathrm{and}\ J^{f'} = \lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \mathrm{at}_P[\![S_f]\!]\ \textbf{?}\ J_{\mathrm{at}_P[\![S_f]\!]} \mathbin{\dot{\sqcup}} \mathrm{Abexp}[\![T(\neg B)]\!](J_\ell)\ \text{¿}\ J_l)\ \mathrm{in} \\
&\qquad \mathrm{let}\ \ J^{t''} = \mathrm{APost}[\![S_t]\!](J^{t'}) \\
&\qquad \mathrm{and}\ J^{f''} = \mathrm{APost}[\![S_f]\!](J^{f'})\ \mathrm{in} \\
&\qquad\quad \lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \ell'\ \textbf{?}\ J^{t''}_{\ell'} \mathbin{\dot{\sqcup}} J^{t''}_{\mathrm{after}_P[\![S_t]\!]} \mathbin{\dot{\sqcup}} J^{f''}_{\ell'} \mathbin{\dot{\sqcup}} J^{f''}_{\mathrm{after}_P[\![S_f]\!]}\ \text{¿}\ J^{t''}_l \mathbin{\dot{\sqcup}} J^{f''}_l)
\end{aligned}$$

$$= \quad \wr\text{by locality (113) and labelling scheme (59) so that in particular } J^{t''}_{\ell'} = J^{t'}_{\ell'} = J^{t}_{\ell'} = J^{f}_{\ell'} = J^{f'}_{\ell'} = J^{f''}_{\ell'} \text{ and APost}[\![S_t]\!] \text{ and APost}[\![S_f]\!] \text{ do not interfere}\wr$$

Simpler checkers?

Proof

$$
\begin{aligned}
& \dot{\ddot{\alpha}}\llbracket P\rrbracket(\mathrm{Post}\llbracket \texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}\rrbracket) \\
= \quad & \wr\text{def. (110) of } \dot{\ddot{\alpha}}\llbracket P\rrbracket \int \\
& \ddot{\alpha}\llbracket P\rrbracket \circ \mathrm{Post}\llbracket \texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}\rrbracket \circ \ddot{\gamma}\llbracket P\rrbracket \\
= \quad & \wr\text{def. (103) of Post}\int \\
& \ddot{\alpha}\llbracket P\rrbracket \circ \mathrm{post}[\tau^\star\llbracket \texttt{if } B \texttt{ then } S_t \texttt{ else } S_f \texttt{ fi}\rrbracket] \circ \ddot{\gamma}\llbracket P\rrbracket \\
= \quad & \wr\text{big step operational semantics (93)}\int \\
& \ddot{\alpha}\llbracket P\rrbracket \circ \mathrm{post}\,[(1_{\Sigma\llbracket P\rrbracket} \cup \tau^B) \circ \tau^\star\llbracket S_t\rrbracket \circ (1_{\Sigma\llbracket P\rrbracket} \cup \tau^t) \cup (1_{\Sigma\llbracket P\rrbracket} \cup \tau^{\bar{B}}) \circ \tau^\star\llbracket S_f\rrbracket \circ (1_{\Sigma\llbracket P\rrbracket} \cup \\
& \tau^f)] \circ \ddot{\gamma}\llbracket P\rrbracket \\
= \quad & \wr\text{Galois connection (98) so that post preserves joins}\int \\
& \ddot{\alpha}\llbracket P\rrbracket \circ (\mathrm{post}\,[(1_{\Sigma\llbracket P\rrbracket} \cup \tau^B) \circ \tau^\star\llbracket S_t\rrbracket \circ (1_{\Sigma\llbracket P\rrbracket} \cup \tau^t)] \mathbin{\dot{\cup}} \\
& \mathrm{post}[(1_{\Sigma\llbracket P\rrbracket} \cup \tau^{\bar{B}}) \circ \tau^\star\llbracket S_f\rrbracket \circ (1_{\Sigma\llbracket P\rrbracket} \cup \tau^f)]) \circ \ddot{\gamma}\llbracket P\rrbracket \\
= \quad & \wr\text{Galois connection (106) so that } \ddot{\alpha}\llbracket P\rrbracket \text{ preserves joins}\int \\
& (\ddot{\alpha}\llbracket P\rrbracket \circ \mathrm{post}[(1_{\Sigma\llbracket P\rrbracket} \cup \tau^B) \circ \tau^\star\llbracket S_t\rrbracket \circ (1_{\Sigma\llbracket P\rrbracket} \cup \tau^t)] \circ \ddot{\gamma}\llbracket P\rrbracket) \mathbin{\dot{\ddot{\sqcup}}} (\ddot{\alpha}\llbracket P\rrbracket \circ \\
& \mathrm{post}[(1_{\Sigma\llbracket P\rrbracket} \cup \tau^{\bar{B}}) \circ \tau^\star\llbracket S_f\rrbracket \circ (1_{\Sigma\llbracket P\rrbracket} \cup \tau^f)] \circ \ddot{\gamma}\llbracket P\rrbracket) \\
\dot{\ddot{\sqsubseteq}} \quad & \wr\text{lemma (5.3) and similar one for the } \texttt{else} \text{ branch}\int \\
& \lambda J \bullet \mathrm{let}\ J^{t'} = \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \bullet (l = \mathrm{at}_P\llbracket S_t\rrbracket \ \mathbf{?}\ J_{\mathrm{at}_P\llbracket S_t\rrbracket} \mathbin{\dot{\sqcup}} \mathrm{Abexp}\llbracket B\rrbracket(J_\ell) \mathrel{\text{¿}} J_l) \ \mathrm{in} \qquad (120) \\
& \qquad \mathrm{let}\ J^{t''} = \mathrm{APost}\llbracket S_t\rrbracket(J^{t'}) \ \mathrm{in} \\
& \qquad\quad \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \bullet (l = \ell' \ \mathbf{?}\ J^{t''}_{\ell'} \mathbin{\dot{\sqcup}} J^{t''}_{\mathrm{after}_P\llbracket S_t\rrbracket} \mathrel{\text{¿}} J^{t''}_l) \\
& \quad \ddot{\sqcup} \\
& \quad \mathrm{let}\ J^{f'} = \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \bullet (l = \mathrm{at}_P\llbracket S_f\rrbracket \ \mathbf{?}\ J_{\mathrm{at}_P\llbracket S_f\rrbracket} \mathbin{\dot{\sqcup}} \mathrm{Abexp}\llbracket T(\neg B)\rrbracket(J_\ell) \mathrel{\text{¿}} J_l) \ \mathrm{in} \\
& \qquad \mathrm{let}\ J^{f''} = \mathrm{APost}\llbracket S_f\rrbracket(J^{f'}) \ \mathrm{in} \\
& \qquad\quad \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \bullet (l = \ell' \ \mathbf{?}\ J^{f''}_{\ell'} \mathbin{\dot{\sqcup}} J^{f''}_{\mathrm{after}_P\llbracket S_f\rrbracket} \mathrel{\text{¿}} J^{f''}_l) \\
= \quad & \wr\text{by grouping similar terms}\int \\
& \lambda J \bullet \mathrm{let}\ J^{t'} = \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \bullet (l = \mathrm{at}_P\llbracket S_t\rrbracket \ \mathbf{?}\ J_{\mathrm{at}_P\llbracket S_t\rrbracket} \mathbin{\dot{\sqcup}} \mathrm{Abexp}\llbracket B\rrbracket(J_\ell) \mathrel{\text{¿}} J_l) \\
& \quad \mathrm{and}\ J^{f'} = \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \bullet (l = \mathrm{at}_P\llbracket S_f\rrbracket \ \mathbf{?}\ J_{\mathrm{at}_P\llbracket S_f\rrbracket} \mathbin{\dot{\sqcup}} \mathrm{Abexp}\llbracket T(\neg B)\rrbracket(J_\ell) \mathrel{\text{¿}} J_l) \ \mathrm{in} \\
& \qquad \mathrm{let}\ J^{t''} = \mathrm{APost}\llbracket S_t\rrbracket(J^{t'}) \\
& \qquad \mathrm{and}\ J^{f''} = \mathrm{APost}\llbracket S_f\rrbracket(J^{f'}) \ \mathrm{in} \\
& \qquad\quad \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \bullet (l = \ell' \ \mathbf{?}\ J^{t''}_{\ell'} \mathbin{\dot{\sqcup}} J^{t''}_{\mathrm{after}_P\llbracket S_t\rrbracket} \mathbin{\dot{\sqcup}} J^{f''}_{\ell'} \mathbin{\dot{\sqcup}} J^{f''}_{\mathrm{after}_P\llbracket S_f\rrbracket} \mathrel{\text{¿}} J^{t''}_l \mathbin{\dot{\sqcup}} J^{f''}_l) \\
= \quad & \wr\text{by locality (113) and labelling scheme (59) so that in particular } J^{t''}_{\ell'} = J^{t'}_{\ell'} = J^{t}_{\ell'} = J^{f}_{\ell'} \\
& \quad = J^{f'}_{\ell'} = J^{f''}_{\ell'} \text{ and APost}\llbracket S_t\rrbracket \text{ and APost}\llbracket S_f\rrbracket \text{ do not interfere}\int
\end{aligned}
$$

Implementation

```
matrix_t* _matrix_alloc_int(const int mr, const int nc)
{
  matrix_t* mat = (matrix_t*)malloc(sizeof(matrix_t));
  mat->nbrows = mat->_maxrows = mr;
  mat->nbcolumns = nc;
  mat->_sorted = s;
  if (mr*nc>0){
    int i;
    pkint_t* q;
    mat->_pinit = _vector_alloc_int(mr*nc);
    mat->p = (pkint_t**)malloc(mr * sizeof(pkint_t*));
    q = mat->_pinit;
    for (i=0;i<mr;i++){
      mat->p[i]=q;
      q=q+nc;
    }}
  return mat;
}

void backsubstitute(matrix_t* con, int rank)
{
  int i,j,k;
  for (k=rank-1; k>=0; k--) {
    j = pk_cherni_intp[k];
    for (i=0; i<k; i++) {
      if (pkint_sgn(con->p[i][j]))
        matrix_combine_rows(con,i,k,i,j);
    }
    for (i=k+1; i<con->nbrows; i++) {
      if (pkint_sgn(con->p[i][j]))
        matrix_combine_rows(con,i,k,i,j);
    }}
}
```

Simpler checkers?

Proof

$\dot{\ddot{\alpha}}[\![P]\!](\mathrm{Post}[\![\mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}]\!])$

$=\quad \wr$def. (110) of $\dot{\ddot{\alpha}}[\![P]\!]\wr$

$\ddot{\alpha}[\![P]\!] \circ \mathrm{Post}[\![\mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}]\!] \circ \ddot{\gamma}[\![P]\!]$

$=\quad \wr$def. (103) of Post$\wr$

$\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[\tau^\star[\![\mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}]\!]] \circ \ddot{\gamma}[\![P]\!]$

$=\quad \wr$big step operational semantics (93)$\wr$

$\ddot{\alpha}[\![P]\!] \circ \mathrm{post}\,[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t) \cup (1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)] \circ \ddot{\gamma}[\![P]\!]$

$=\quad \wr$Galois connection (98) so that post preserves joins$\wr$

$\ddot{\alpha}[\![P]\!] \circ (\mathrm{post}\,[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t)] \,\dot{\cup}\, \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)]) \circ \ddot{\gamma}[\![P]\!]$

$=\quad \wr$Galois connection (106) so that $\ddot{\alpha}[\![P]\!]$ preserves joins$\wr$

$(\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^B) \circ \tau^\star[\![S_t]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^t)] \circ \ddot{\gamma}[\![P]\!]) \,\dot{\ddot{\sqcup}}\, (\ddot{\alpha}[\![P]\!] \circ \mathrm{post}[(1_{\Sigma[\![P]\!]} \cup \tau^{\bar{B}}) \circ \tau^\star[\![S_f]\!] \circ (1_{\Sigma[\![P]\!]} \cup \tau^f)] \circ \ddot{\gamma}[\![P]\!])$

$\dot{\ddot{\sqsubseteq}}\quad \wr$le

$\lambda J \bullet$ let

le

$\lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \ell'$? $J_{\ell'} \,\dot{\sqcup}\, J_{\mathrm{after}_P[\![S_t]\!]}$ ¿ J_l)

$\ddot{\sqcup}$

let $J^{f'} = \lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \mathrm{at}_P[\![S_f]\!]$? $J_{\mathrm{at}_P[\![S_f]\!]} \,\dot{\sqcup}\, \mathrm{Abexp}[\![T(\neg B)]\!](J_\ell)$ ¿ J_l) in

let $J^{f''} = \mathrm{APost}[\![S_f]\!](J^{f'})$ in

$\lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \ell'$? $J^{f''}_{\ell'} \,\dot{\sqcup}\, J^{f''}_{\mathrm{after}_P[\![S_f]\!]}$ ¿ $J^{f''}_l$)

$=\quad \wr$by grouping similar terms$\wr$

$\lambda J \bullet$ let $J^{t'} = \lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \mathrm{at}_P[\![S_t]\!]$? $J_{\mathrm{at}_P[\![S_t]\!]} \,\dot{\sqcup}\, \mathrm{Abexp}[\![B]\!](J_\ell)$ ¿ J_l)

and $J^{f'} = \lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \mathrm{at}_P[\![S_f]\!]$? $J_{\mathrm{at}_P[\![S_f]\!]} \,\dot{\sqcup}\, \mathrm{Abexp}[\![T(\neg B)]\!](J_\ell)$ ¿ J_l) in

let $J^{t''} = \mathrm{APost}[\![S_t]\!](J^{t'})$

and $J^{f''} = \mathrm{APost}[\![S_f]\!](J^{f'})$ in

$\lambda l \in \mathrm{in}_P[\![P]\!] \bullet (l = \ell'$? $J^{t''}_{\ell'} \,\dot{\sqcup}\, J^{t''}_{\mathrm{after}_P[\![S_t]\!]} \,\dot{\sqcup}\, J^{f''}_{\ell'} \,\dot{\sqcup}\, J^{f''}_{\mathrm{after}_P[\![S_f]\!]}$ ¿ $J^{t''}_l \,\dot{\sqcup}\, J^{f''}_l$)

$=\quad \wr$by locality (113) and labelling scheme (59) so that in particular $J^{t''}_{\ell'} = J^{t'}_{\ell'} = J^{t}_{\ell'} = J^{f}_{\ell'} = J^{f'}_{\ell'} = J^{f''}_{\ell'}$ and $\mathrm{APost}[\![S_t]\!]$ and $\mathrm{APost}[\![S_f]\!]$ do not interfere$\wr$

Implementation

```
matrix_t* _matrix_alloc_int(const int mr, const int nc)
{
  matrix_t* mat = (matrix_t*)malloc(sizeof(matrix_t));
  mat->nbrows = mat->_maxrows = mr;
  mat->nbcolumns = nc;
  mat->_sorted = s;
  if (mr*nc>0){
    int i;
    pkint_t* q;
    mat->_pinit = _vector_alloc_int(mr*nc);
    mat->p = (pkint_t**)malloc(mr * sizeof(pkint_t*));
    q = mat->_pinit;
    for (i=0;i<mr;i++){
      mat->p[i]=q;
      q=q+nc;
```

```
void backsubstitute(matrix_t* con, int rank)
{
  int i,j,k;
  for (k=rank-1; k>=0; k--) {
    j = pk_cherni_intp[k];
    for (i=0; i<k; i++) {
      if (pkint_sgn(con->p[i][j]))
        matrix_combine_rows(con,i,k,i,j);
    }
    for (i=k+1; i<con->nbrows; i++) {
      if (pkint_sgn(con->p[i][j]))
        matrix_combine_rows(con,i,k,i,j);
    }}
}
```

Do the two parts connect?

Really simple checkers?

Bytecode verification (together with stack inspection) is the basis of Java security.

- Dataflow analysis ensures that values are manipulated with correct types, methods are applied to correct arguments, no stack underflows and overflows. . .
- Preceeded by a structural analysis that ensures that the code is well-formed and methods, names, and classes exist. . .
- and that jumps remain with code!
- In 2004, Godwiak exploited failure of BCV to verify targets of jumps to launch attacks on Nokia phones
- No verifier for a real language is really simple!

Really simple checkers?

Bytecode verification (together with stack inspection) is the basis of Java security.

- Dataflow analysis ensures that values are manipulated with correct types, methods are applied to correct arguments, no stack underflows and overflows...
- Preceeded by a structural analysis that ensures that the code is well-formed and methods, names, and classes exist...
- and that jumps remain with code!
- In 2004, Godwiak exploited failure of BCV to verify targets of jumps to launch attacks on Nokia phones
- No verifier for a real language is really simple!

Really simple checkers?

Bytecode verification (together with stack inspection) is the basis of Java security.

- Dataflow analysis ensures that values are manipulated with correct types, methods are applied to correct arguments, no stack underflows and overflows. . .
- Preceeded by a structural analysis that ensures that the code is well-formed and methods, names, and classes exist. . .
- and that jumps remain with code!
- In 2004, Godwiak exploited failure of BCV to verify targets of jumps to launch attacks on Nokia phones
- No verifier for a real language is really simple!

Really simple checkers?

Bytecode verification (together with stack inspection) is the basis of Java security.

- Dataflow analysis ensures that values are manipulated with correct types, methods are applied to correct arguments, no stack underflows and overflows. . .
- Preceeded by a structural analysis that ensures that the code is well-formed and methods, names, and classes exist. . .
- and that jumps remain with code!
- In 2004, Godwiak exploited failure of BCV to verify targets of jumps to launch attacks on Nokia phones
- No verifier for a real language is really simple!

Really simple checkers?

Bytecode verification (together with stack inspection) is the basis of Java security.

- Dataflow analysis ensures that values are manipulated with correct types, methods are applied to correct arguments, no stack underflows and overflows. . .
- Preceeded by a structural analysis that ensures that the code is well-formed and methods, names, and classes exist. . .
- and that jumps remain with code!
- In 2004, Godwiak exploited failure of BCV to verify targets of jumps to launch attacks on Nokia phones
- No verifier for a real language is really simple!

Foundational Proof Carrying Code

Theorem

Executions of program p are safe.

- Proof proceeds by showing that safety is an invariant of execution, under assumptions given for p
- depends on the definition of execution.
  - For the JVM: a 400 pages book!
- TCB of Foundational PCC:
  1. the proof checker (as before)
  2. the formal definition of the language semantics
  3. the formal definition of the policy
- This is also a large TCB
- Still better to have 2,000 lines of formal definitions than with 20,000 lines of C code!

Foundational Proof Carrying Code

Theorem

Executions of program p are safe.

- Proof proceeds by showing that safety is an invariant of execution, under assumptions given for p
- depends on the definition of execution.
  - For the JVM: a 400 pages book!
- TCB of Foundational PCC:
  1. the proof checker (as before)
  2. the formal definition of the language semantics
  3. the formal definition of the policy
- This is also a large TCB
- Still better to have 2,000 lines of formal definitions than with 20,000 lines of C code!

Foundational Proof Carrying Code

Theorem

Executions of program p are safe.

- Proof proceeds by showing that safety is an invariant of execution, under assumptions given for p
- depends on the definition of execution.
  - For the JVM: a 400 pages book!
- TCB of Foundational PCC:
  1. the proof checker (as before)
  2. the formal definition of the language semantics
  3. the formal definition of the policy
- This is also a large TCB
- Still better to have 2,000 lines of formal definitions than with 20,000 lines of C code!

Foundational Proof Carrying Code

Theorem

Executions of program p are safe.

- Proof proceeds by showing that safety is an invariant of execution, under assumptions given for p
- depends on the definition of execution.
  - For the JVM: a 400 pages book!
- TCB of Foundational PCC:
  1. the proof checker (as before)
  2. the formal definition of the language semantics
  3. the formal definition of the policy
- This is also a large TCB
- Still better to have 2,000 lines of formal definitions than with 20,000 lines of C code!

Foundational Proof Carrying Code

Theorem

Executions of program p are safe.

- Proof proceeds by showing that safety is an invariant of execution, under assumptions given for p
- depends on the definition of execution.
    - For the JVM: a 400 pages book!
- TCB of Foundational PCC:
    1. the proof checker (as before)
    2. the formal definition of the language semantics
    3. the formal definition of the policy
- This is also a large TCB
- Still better to have 2,000 lines of formal definitions than with 20,000 lines of C code!

Executable checkers

- In foundational PCC, certificates represent deductive proofs
  - Typing rules as lemmas
- A better alternative is to program a type system/VCGen in the proof checker and prove it correct!
  - Scalable and shorter proof terms
  - Allows extraction of certified checkers

Executable checkers vs Foundational PCC

Reflection

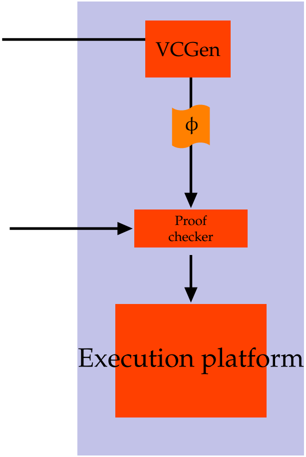
Use computations instead of deductions!

- A predicate $P : T \rightarrow \texttt{Prop}$
- A decision procedure $f : T \rightarrow \mathrm{bool}$
- A correctness lemma $C : \forall x : T.\, f\ x = \mathrm{true} \rightarrow P\ x$

If $f\ a$ reduces to true, then $C\ a\ (\texttt{refl\_eq}\ \mathrm{true})$ is a proof of $P\ a$

- Executable checkers provide the same guarantees than FPCC
- Executable checkers can be seen as efficient procedures to generate compact certificates

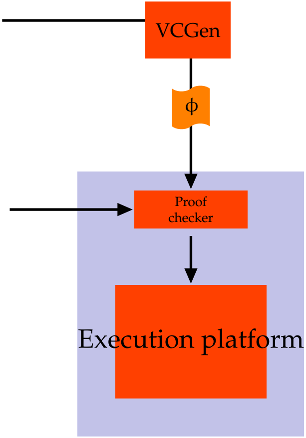
TCB of certified PCC



1. In standard PCC
2. If the VCGen is proved correct
   - \+ the proof checker
   - \+ the formal definition of the language semantics
   - \+ the formal definition of the policy

   (same as FPCC)

TCB of certified PCC

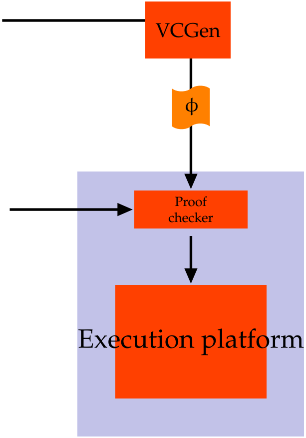


1. In standard PCC
2. If the VCGen is proved correct

    + the proof checker
    + the formal definition of the language semantics
    + the formal definition of the policy

(same as FPCC)

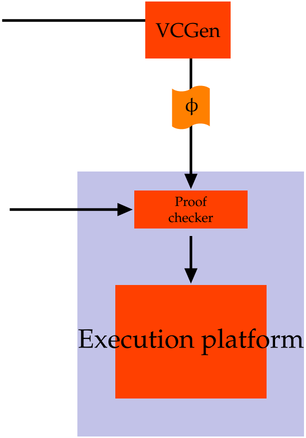
TCB of certified PCC



1. In standard PCC
2. If the VCGen is proved correct
   + the proof checker
   + the formal definition of the language semantics
   + the formal definition of the policy

(same as FPCC)

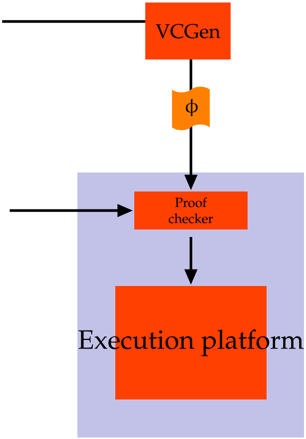
TCB of certified PCC



1. In standard PCC
2. If the VCGen is proved correct
   - + the proof checker
   - + the formal definition of the language semantics
   - + the formal definition of the policy

   (same as FPCC)

TCB of certified PCC



1. In standard PCC
2. If the VCGen is proved correct
   - + the proof checker
   - + the formal definition of the language semantics
   - + the formal definition of the policy

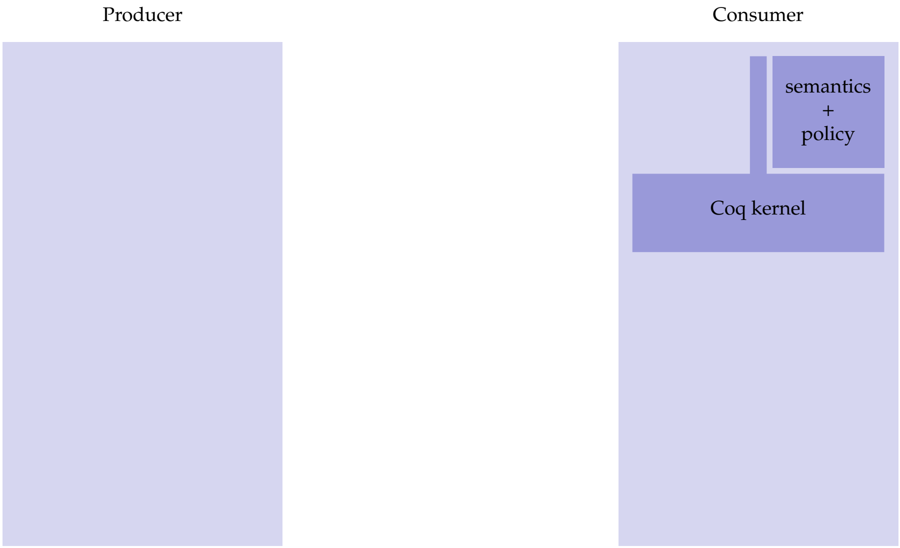
   (same as FPCC)

Using executable checkers

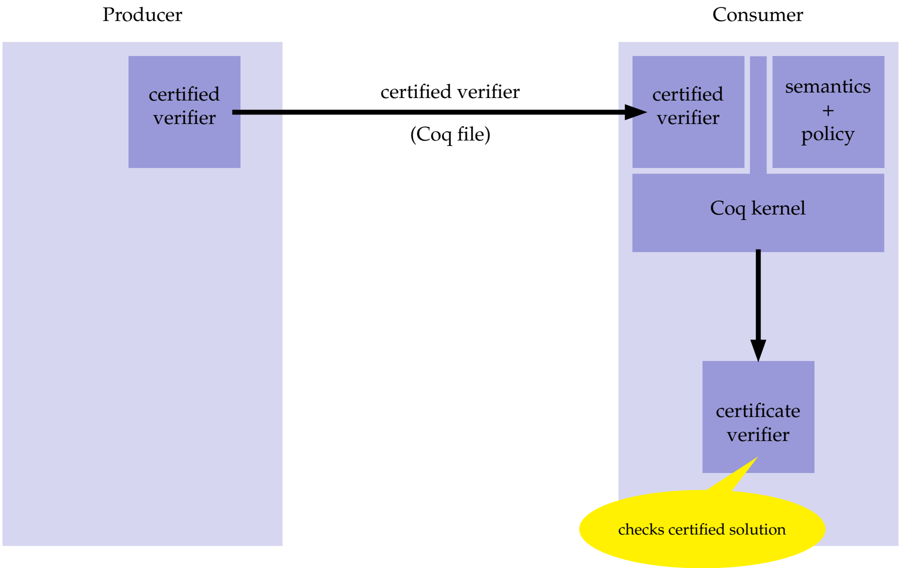
Producer

Consumer

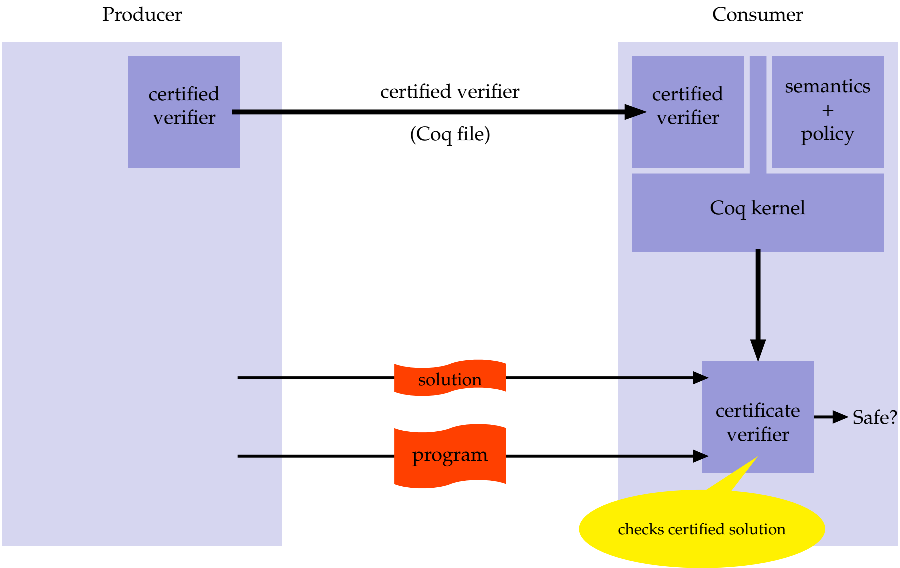
Using executable checkers



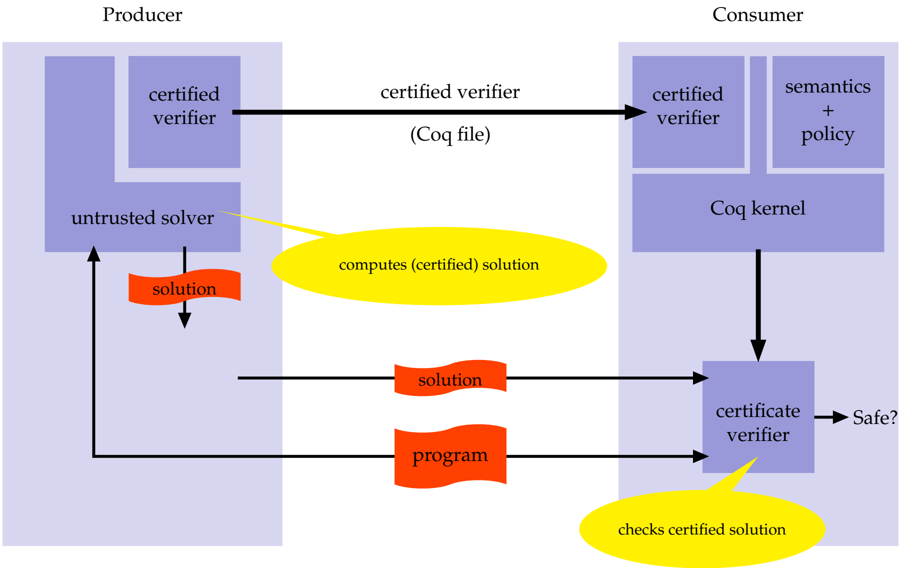
Using executable checkers



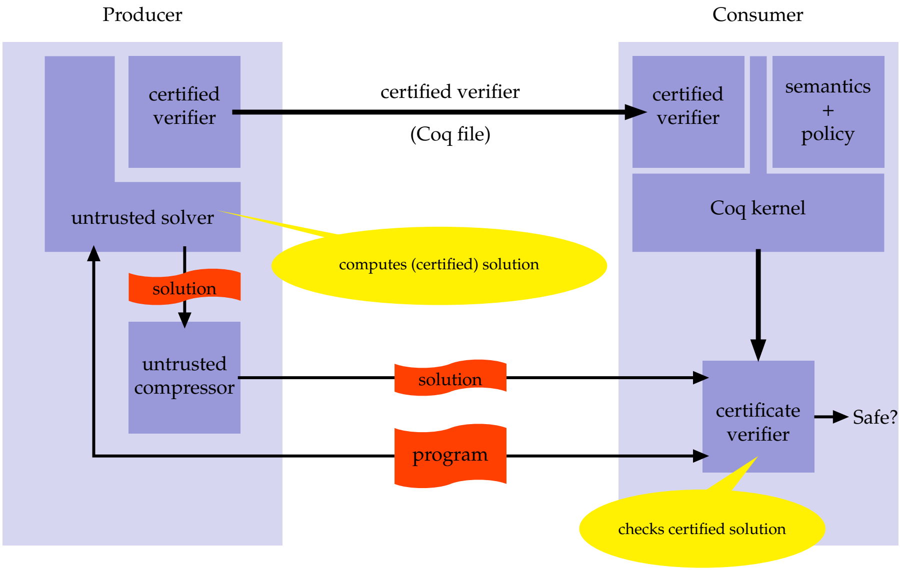
Using executable checkers



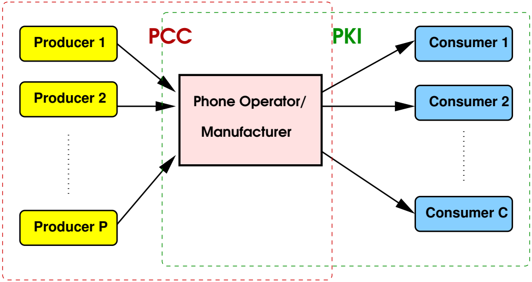
Using executable checkers



Using executable checkers

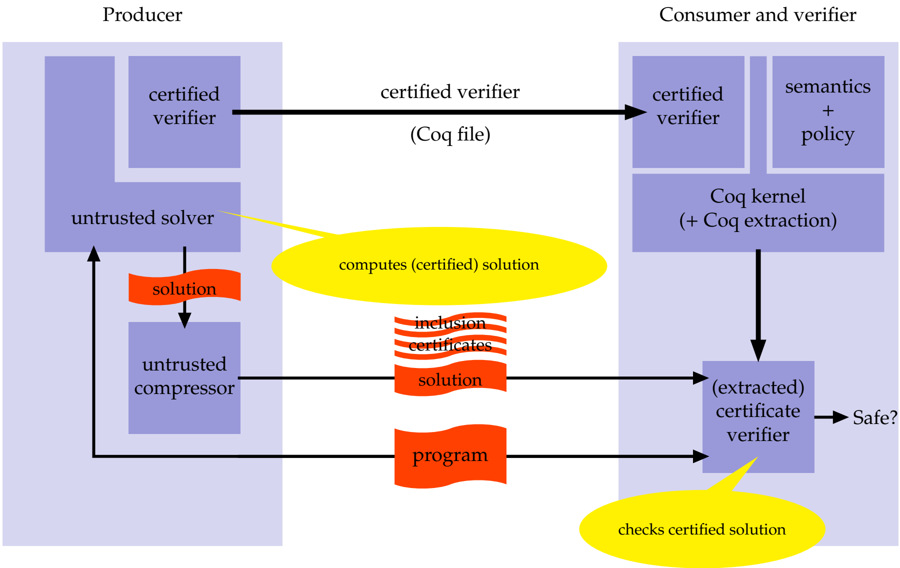


Application scenario: PCC with trusted intermediaries



- Size of certificate not a major issue
- Can check whether certified policy meets expected policy
- Complex policies can be verified

Using executable checkers



Application scenario: retail PCC

- Trusted intermediary validates verifier
- User validates application
- Size of certificate an issue
- Restricted to simpler policies
- Increased flexibility

Objectives

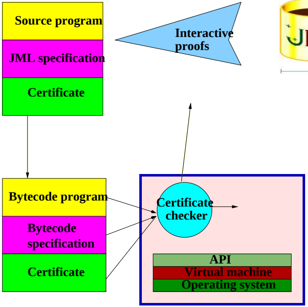
Present two instances of **certified Proof Carrying code** and provide methods to generate certificates from **source code verification**

- Type system for information flow based confidentiality policies
- Verification condition generator for logical specifications

Objectives

Present two instances of **certified Proof Carrying code** and provide methods to generate certificates from **source code verification**

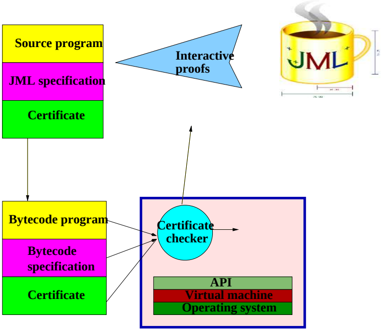
- Type system for information flow based confidentiality policies
- Verification condition generator for logical specifications

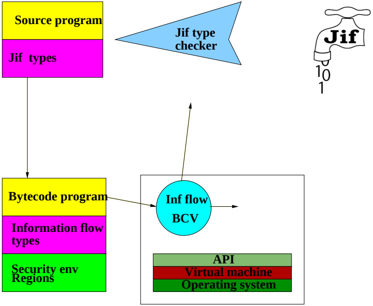


Objectives

Present two instances of **certified Proof Carrying code** and provide methods to generate certificates from **source code verification**

- Type system for information flow based confidentiality policies
- Verification condition generator for logical specifications





Proof assistants based on type theory

Type theory is a language for:

- defining mathematical objects (including data structures, algorithms, and mathematical theories)
- performing computations on and with these objects
- reasoning about these objects

It is a foundational language that underlies:

- proof assistants (inc. Coq, Epigram, Agda)
- programming languages (inc. Cayenne, DML).

Proof assistants

- Implement type theories/higher order logics to specify and reason about mathematics.
- Interactive proofs, with mechanisms to guarantee that
  - theorems are applied with the right hypotheses
  - functions are applied to the right arguments
  - no missing cases in proofs or in function definitions
  - no illicit logical step (all reasoning is reduced to elementary steps)

  Proof assistants include domain-specific tactics that help solving specific problems efficiently.

Proof objects as certificates

- Completed proofs are represented by proof objects that can easily be checked by a proof-checker.
- Proof checker is small.

Sample applications (many more)

- Programming languages
  - Programming language semantics
  - Program transformations: compilers, partial evaluators, normalizers
  - Program verification: type systems, Hoare logics, verification condition generators,
- Operating systems
- Cryptographic protocols and algorithms
  - Dolev-Yao model (perfect cryptography assumption)
  - Computational model
- Mathematics and logic:
  - Galois theory, category theory, real numbers, polynomials, computer algebra systems, geometry, group theory, etc.
  - 4-colors theorem
  - Type theory

Type theory and the Curry-Howard isomorphism

- Type theory is a programming language for writing algorithms.
  - But all functions are total and terminating, so that convertibility is decidable.
- Type theory is a language for proofs, via the Curry-Howard isomorphism:

Propositions	=	Types
Proofs	=	Terms
Proof-Checking	=	Type-Checking

  - But the underlying logic is constructive. (Classical logic can be recovered with an axiom, or a control operator)

A Theory of Functions

- Judgements

$$x_1 : A_1, \ \ldots \ , x_n : A_n \vdash M : B$$

- Typing rules

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \to B}$$

- Evaluation: computing the application a function to an argument

$$(\lambda x : A.\, M)\; N \quad \to_\beta \quad M\{x := N\}$$

  - The result of computation is unique

$$M =_\beta N \quad \Rightarrow \quad M \downarrow_\beta N$$

  - Evaluation preserves typing
- Type-Checking: it is decidable whether $\Gamma \vdash M : A$.
- Type-Inference: there exists a partial function inf s.t.

$$\Gamma \vdash M : A \quad \Leftrightarrow \quad \Gamma \vdash M : (\mathsf{inf}(\Gamma, M)) \wedge (\mathsf{inf}(\Gamma, M)) = A$$

A Language for Proofs

Minimal Intuitionistic Logic

- Formulae:
$$\begin{array}{rcl} \mathfrak{F} & = & \mathfrak{X} \\ & \mid & \mathfrak{F} \to \mathfrak{F} \end{array}$$
- Judgements
$A_1, \ \ldots \ , A_n \vdash B$
- Derivation rules

$$\frac{}{\Gamma \ \vdash \ A} \quad A \in \Gamma$$

$$\frac{\Gamma \ \vdash \ A \to B \qquad \Gamma \ \vdash \ A}{\Gamma \ \vdash \ B}$$

$$\frac{\Gamma, A \ \vdash \ B}{\Gamma \ \vdash \ A \to B}$$

- If $\Gamma \ \vdash \ M : A$ then $\Gamma \ \vdash \ A$
- If $\Gamma \ \vdash \ A$ then $\Gamma \ \vdash \ M : A$ for some M
- (A tight correspondence between derivation trees and λ-terms, and between proof normalization and β-reduction)
- In a proof assistant M is often built backwards.

BHK Interpretation

A proof of:	is given by:
$A \wedge B$	a proof of A and a proof of B
$A \vee B$	a proof of A or a proof of B
$A \to B$	a method to transform proofs of A into proofs of B
$\forall x.\ A$	a method to produce a proof of $A(t)$ for every t
$\exists x.\ A$	a witness t and a proof of $A(t)$
$\bot$	has no proof

Use dependent types (terms arise in types) to achieve the expressive power of predicate logics

$$\begin{array}{ll} & N : \mathbf{Type}, O : N, P : N \to \mathbf{Prop} \\ \vdash & \lambda x :\ (P\ O).\ x : (P\ O) \to P((\lambda z : N.\ z)\ O) \end{array}$$

Typing dependent types: Calculus of Constructions

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.\, B) : s_2} \qquad (s_1, s_2) \in \mathcal{R}$$

$$\frac{\Gamma \vdash F : (\Pi x{:}A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B\{x := a\}}$$

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.\, B) : s}{\Gamma \vdash \lambda x{:}A.\, b : \Pi x{:}A.\, B}$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \qquad B =_\beta B'$$

Rules

- $(\mathtt{Prop}, \mathtt{Prop})$ implication
- $(\mathtt{Type}, \mathtt{Type})$ generalized function space
- $(\mathtt{Type}, \mathtt{Prop})$ universal quantification
- $(\mathtt{Prop}, \mathtt{Type})$ precondition, etc

Inductive definitions

- Inductive definitions provide mechanisms to define data structures, to define recursive functions and to reason about inhabitants of data structures
  - recursors/case-expressions and guarded fixpoints/pattern matching
  - induction principles
- Encode a rich class of structures:
  - algebraic types: booleans, binary natural numbers, integers, etc
  - parameterized types: lists, trees, etc
  - inductive families and relations: vectors, accessibility relations (to define functions by well-founded recursion), transition systems, etc.
- Extensively used in the formalization of mathematics, programming languages, cryptographic algorithms, in reflexive tactics, etc.

Typing rules for natural numbers

$$\vdash \ \mathsf{Nat} : s \qquad \vdash \ 0 : \mathsf{Nat} \qquad \frac{\Gamma \ \vdash \ n : \mathsf{Nat}}{\Gamma \ \vdash \ S\ n : \mathsf{Nat}}$$

$$\frac{\Gamma \ \vdash \ n : \mathsf{Nat} \quad \Gamma \ \vdash \ f_0 : A \quad \Gamma \ \vdash \ f_s : \mathsf{Nat} \to A}{\Gamma \ \vdash \ \mathsf{case}\ n\ \mathsf{of}\{0 \Rightarrow f_0 \mid s \Rightarrow f_s\} : A}$$

$$\frac{\begin{array}{c}\Gamma \ \vdash \ n : \mathsf{Nat} \quad \Gamma \ \vdash \ P : \mathsf{Nat} \to s \\ \Gamma \ \vdash \ f_0 : P\ 0 \quad \Gamma \ \vdash \ f_s : \Pi n{:}\mathsf{Nat}.\ P\ (S\ n)\end{array}}{\Gamma \ \vdash \ \mathsf{case}\ n\ \mathsf{of}\{0 \Rightarrow f_0 \mid s \Rightarrow f_s\} : P\ n}$$

$$\frac{\Gamma, f : \mathsf{Nat} \to A \ \vdash \ e : \mathsf{Nat} \to A}{\Gamma \ \vdash \ \mathsf{letrec}\ f = e : \mathsf{Nat} \to A}$$

Case expressions and fixpoints: reduction rules

$$\begin{array}{rcl}
\mathsf{case}\ 0\ \mathsf{of}\{0 \Rightarrow e_0 \mid s \Rightarrow e_s\} & \to & e_0 \\
\mathsf{case}\ (s\ n)\ \mathsf{of}\{0 \Rightarrow e_0 \mid s \Rightarrow e_s\} & \to & e_s\ n \\
(\mathsf{letrec}\ f = e)\ n & \to & e\{f := (\mathsf{letrec}\ f = e)\}\ n
\end{array}$$

To ensure termination

- we use a side condition $\mathcal{G}(f, e)$, read f is guarded in e, in the typing rule for fixpoint
- we require n to be of the form $c\ \vec{b}$ in the reduction rule in the reduction rule for fixpoint

Not sufficient to impose restrictions on fixpoint definitions. Must also guarantee inductive definitions are well-formed.

Example: formalizing semantics of expressions

$$
\begin{array}{lcl}
a & \in & \mathbf{AExp} \\
\\
a & := & n \\
  & \mid & x \\
  & \mid & a_1 + a_2 \\
  & \mid & a_1 - a_2 \\
  & \mid & a_1 * a_2
\end{array}
\qquad
\begin{array}{lcl}
b & \in & \mathbf{BExp} \\
\\
b & := & \text{true} \\
  & \mid & \text{false} \\
  & \mid & a_1 = a_2 \\
  & \mid & a_1 < a_2 \\
  & \mid & \text{not } b \\
  & \mid & b_1 \text{ and } b_2
\end{array}
\qquad
\begin{array}{lcl}
c & \in & \mathbf{Comm} \\
\\
c & := & \text{skip} \\
  & \mid & x := a \\
  & \mid & c_1 ; c_2 \\
  & \mid & \text{if } b \text{ then } c_1 \text{ else } c_2 \\
  & \mid & \text{while } b \text{ do } c
\end{array}
$$

Shallow embedding

- Expressions have type $\mathsf{mem} \to \mathsf{Nat}$
- Memories have type $\mathsf{mem} = \mathsf{loc} \to \mathsf{Nat}$

$$
\begin{array}{l}
\mathsf{Num}[v\colon \mathsf{Nat}] = \lambda s{:}\mathsf{mem}.\ v \\
\mathsf{Loc}[v\colon \mathsf{loc}] = \lambda s{:}\mathsf{mem}.\ s\ v \\
\mathsf{Plus}[e1, e2\colon \mathsf{Exp}] = \lambda s{:}\mathsf{mem}.\ (e1\ s) + (e2\ s) \\
\mathsf{Minus}[e1, e2\colon \mathsf{Exp}] = \lambda s{:}\mathsf{mem}.\ (e1\ s) - (e2\ s) \\
\mathsf{Mult}[e1, e2\colon \mathsf{Exp}] = \lambda s{:}\mathsf{mem}.\ (e1\ s) * (e2\ s)
\end{array}
$$

$$x, y\colon \mathsf{Exp} \vdash \mathsf{Plus}\ x\ (\mathsf{Minus}\ y\ (\mathsf{Num}\ 3))\colon \mathsf{Exp}$$

- Expressions of the object language are (undistinguished) terms of the specification language
- Expressions are evaluated using the evaluation system of underlying specification language
- Cannot talk about expressions of the object language

Deep embedding

- Represent explicitely the syntax of the object language
- Possible to compute and reason about expressions of the object language
- Explicit function *eval* needed to evaluate terms

```
Inductive aExp : Set :=
   Loc: loc ->  aExp
 | Num: nat ->  aExp
 | Plus: aExp -> aExp ->  aExp
 | Minus: aExp -> aExp ->  aExp
 | Mult: aExp -> aExp ->  aExp .
```

```
Inductive bExp : Set :=
   IMPtrue: bExp
 | IMPfalse: bExp
 | Equal: aExp -> aExp ->  bExp
 | LessEqual: aExp -> aExp ->  bExp
 | Not: bExp ->  bExp
 | Or: bExp -> bExp ->  bExp
 | And: bExp -> bExp ->  bExp .
```

```
Inductive com : Set :=
   Skip: com
 | Assign: loc -> aExp ->  com
 | Scolon: com -> com ->  com
 | IfThenElse: bExp -> com -> com ->  com
 | WhileDo: bExp -> com ->  com .
```

Semantics of arithmetic expressions: inductive style

Memory mem = loc → Nat

Evaluation relation $\langle a, \sigma \rangle \rightarrow^{\mathsf{a}} n$, i.e. $\rightarrow^{\mathsf{a}} \subseteq \mathbf{AExp} \times \Sigma \times \mathbb{N}$

Evaluation rules

$$\langle n, \sigma \rangle \rightarrow^{\mathsf{a}} n \qquad \langle x, \sigma \rangle \rightarrow^{\mathsf{a}} \sigma(x) \qquad \frac{\langle a_1, \sigma \rangle \rightarrow^{\mathsf{a}} n_1 \quad \langle a_2, \sigma \rangle \rightarrow^{\mathsf{a}} n_2}{\langle a_1 + a_2, \sigma \rangle \rightarrow^{\mathsf{a}} n_1 \underline{+} n_2}$$

```
Inductive evalaExp_ind : aExp -> memory -> nat ->  Prop :=
  eval_Loc: forall (v:locs)(n:nat)(s : memory),
     (lookup s v)=n -> (evalaExp_ind (Loc v) s n)

 | eval_Num: forall (n : nat) (s : memory),
     (evalaExp_ind (Num n) s n)

 | eval_Plus: forall (a0, a1 : aExp) (n0, n1, n : nat) (s : memory),
     (evalaExp_ind a0 s n0) ->
     (evalaExp_ind a1 s n1) ->
     n = (plus n0 n1) ->  (evalaExp_ind (Plus a0 a1) s n)
...
```

Semantics of arithmetic expressions – functional style

```
Fixpoint evalaExp_rec [a: aExp] : memory ->  nat :=
 fun (s : memory) =>
  match a with
     (Loc v) => (lookup s v)
   | (Num n) => n
   | (Plus a1 a2) => (plus (evalaExp_rec a1 s) (evalaExp_rec a2 s))
   | ...
  end.
```

Possible difficulties with functional semantics

- Determinacy
- Partiality
- Termination

For commands:

- Small-step semantics is possible to define but
  - many undefined cases to handle
  - still harder to reason about than inductive semantics
- Big-step semantics is hard (requires well-founded recursion)

Certifying type-based methods

- Bytecode verification
- Abstraction-carrying code
- Non-interference

Bytecode verification: goals

Bytecode verification aims to contribute to safe execution of programs by enforcing:

- Values are used with the right types
  (no pointer arithmetic)
- Operand stack is of appropriate length
  (no overflow, no underflow)
- Subroutines are correct
- Object initialization

But well-typed programs do not go wrong

(With some limits: array bound checks, interfaces, etc)

Bytecode verification: principles

- Exhibit for each program point an abstraction of the local variables and of the operand stack, and verify that instructions are compatible with the abstraction

Informally

$\vdash iadd : (rt, \text{int} :: \text{int} :: s) \Rightarrow (rt, \text{int} :: s)$ $\quad \nvdash iadd : (rt, \text{bool} :: \text{int} :: s) \Rightarrow (rt, \text{int} :: s)$

$\vdash pop : (rt, \alpha :: s) \Rightarrow (rt, s)$ $\quad \nvdash pop : (rt, s) \Rightarrow (rt, s)$

- Compatibility w.r.t. stack types is formalized by transfer rules

$$\frac{P[i] = ins}{i \vdash lv, st \Rightarrow lv', st'} \qquad \frac{P[i] = ins}{i \vdash lv, st \Rightarrow}$$

- Program $P : \tau$ is type-safe if there exists $S : \mathcal{P} \rightarrow \mathcal{RT} \times \mathcal{T}^\star$ s.t.
  - $S_1 = (rt_1, \epsilon)$
  - for all $i, j \in \mathcal{P}$
    - $i \mapsto j \Rightarrow \exists \sigma.\, i \vdash S_i \Rightarrow \sigma \sqsubseteq S_j$;
    - $i \mapsto \Rightarrow \exists \tau'.\, i \vdash S_i \Rightarrow \tau' \sqsubseteq \tau$

  where $\sqsubseteq$ is inherited from JVM types

Bytecode verification: consequences

Programs do not go wrong

If $S \vdash P : \tau$ and s is type-correct w.r.t. S_i and Γ, then:

- $P[i] = \mathsf{return}$ then the return value has type τ
- $s \rightsquigarrow s'$ and s' is type-correct w.r.t. $S_{i'}$ (where $i = pc(s)$ and $i' = pc(s')$)

Run-time type checking is redundant

- A typed state is a state that manipulates typed values (instead of untyped values)
- A defensive virtual machine checks types at execution, i.e. $\rightsquigarrow_{\mathrm{def}} \subseteq \mathsf{tstate} \times (\mathsf{tstate} + \{\mathbf{TypeError}\})$
- If P is type-safe w.r.t. S, then executions of $\rightsquigarrow$ and $\rightsquigarrow_{\mathrm{def}}$ coincide

Type inference

Goal is to exhibit S.

- Entry point of program is typed with the empty stack
- Propagation
  - Pick an program point i annotated with st
  - Compute rt', st' such that $i \vdash rt, st \Rightarrow rt', st'$.
    - If there is no rt', st', then reject program.
  - For all successors j of i
    - if j is not yet annotated, annotated it with rt', st'
    - if j is annotated with rt'', st'', replace rt'', st'' by $rt', st' \sqcup rt'', st''$
  - Upon termination
    - accept program if no type error $\top$ in the computed S.
- Termination is ensured by
  - tracking which states remain to be analyzed,
  - by ascending chain condition

Fixpoint computation!

Lightweight bytecode verification

Provide types of junction points

- Entry point and junction points are typed
  - the entry point of the program is typed with the empty stack
- Propagation
  - Pick an program point i annotated with st
  - Compute rt', st' such that $i \vdash rt, st \Rightarrow rt', st'$. If there is no rt', st', then reject program.
  - For all successors j of i
    - if j is not yet annotated, annotated it with rt', st'
    - if j is annotated with rt'', st'', check that $(rt', st') \sqsubseteq (rt'', st'')$. If not, reject program

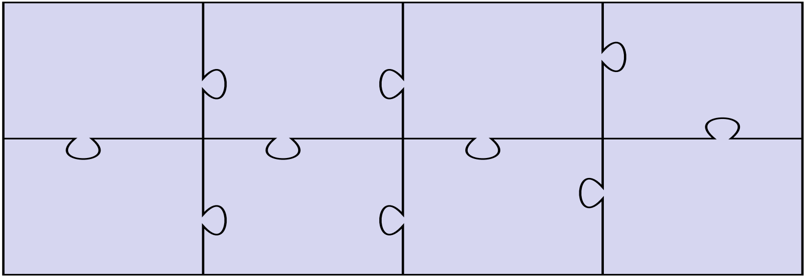
Lightweight bytecode verification

Provide types of junction points

- Entry point and junction points are typed
  - the entry point of the program is typed with the empty stack
- Propagation
  - Pick an program point i annotated with st
  - Compute rt', st' such that $i \vdash rt, st \Rightarrow rt', st'$. If there is no rt', st', then reject program.
  - For all successors j of i
    - if j is not yet annotated, annotated it with rt', st'
    - if j is annotated with rt'', st'', check that $(rt', st') \sqsubseteq (rt'', st'')$. If not, reject program

One pass verification, sound and complete wrt bytecode verification

Verified bytecode verification



- A puzzle with 8 pieces,
- Each piece interacts with its neighbors

Bicolano

- a Coq formalisation of the JVM
- the basis for certified PCC

Initially a joint work effort between INRIA Sophia-Antipolis and IRISA, now developed/used by many other sites

Initial requirements

- a *direct* translation of the reference book,
- readable (even for non Coq expert),
- easy to manipulate in proofs,
- support executable checkers,
- avoid implementation choices

Bicolano vs requirements

Bicolano should be

- a *direct* translation of the reference book,
    -
- readable (even for non Coq expert),
    -
- easy to manipulate in proofs,
    -
- support executable checkers
    -

Bicolano vs requirements

Bicolano should be

- a *direct* translation of the reference book,
  - small step semantics, same level of details (not a JVM implementation)
- readable (even for non Coq expert),
  -
- easy to manipulate in proofs,
  -
- support executable checkers
  -

Bicolano vs requirements

Bicolano should be

- a *direct* translation of the reference book,
  - small step semantics, same level of details (not a JVM implementation)
- readable (even for non Coq expert),
  - use of module interfaces
- easy to manipulate in proofs,
  -
- support executable checkers
  -

Bicolano vs requirements

Bicolano should be

- a *direct* translation of the reference book,
  - small step semantics, same level of details (not a JVM implementation)
- readable (even for non Coq expert),
  - use of module interfaces
- easy to manipulate in proofs,
  - inductive definitions
- support executable checkers
  -

Bicolano vs requirements

Bicolano should be

- a *direct* translation of the reference book,
  - small step semantics, same level of details (not a JVM implementation)
- readable (even for non Coq expert),
  - use of module interfaces
- easy to manipulate in proofs,
  - inductive definitions
- support executable checkers
  - implementation of module interfaces

Java fragment handled

- numeric values : int, short, byte
  - no float, no double, no long
  - no 64 bits values: complex management of 64 and 32 bits elements in the operand stack
- objects, arrays
- virtual method calls
  - class hierarchy is dynamically traversed to find a suitable implementation
- visibility modifiers
- exceptions
- programs are post-linked
  (no constant pool, no dynamical linking)
- no initialisation (use default values instead)
- no subroutines (CLDC!)

Syntax

Factorisation:

- Binary operations on int: `ibinop` *op*
  (`iadd` ,`iand` ,`idiv` ,`imul` ,`ior` ,`irem` ,`ishl` ,`ishr` ,`isub` ,`iushr` ,`ixor`)
- Tests on int value : `if0` *comp*
  (`ifeq` ,`ifne` ,`iflt` ,`ifle` ,`ifgt` ,`ifge`)
- Push numerical constants on the operand stack: `const` t c
  (`bipush`, `iconst_<i>`, `ldc`, `sipush`)
- load value from local variables : `aload`, `iload`
- load value from array : `aaload`, `baload`, `iaload`, `saload`
- similar instructions to store values...

Wellformedness properties on programs

Some examples

- all the classes have a super-class except `java.lang.Object`,
- the class hierarchy is not cyclic,
- all class have distinct names,
- ...

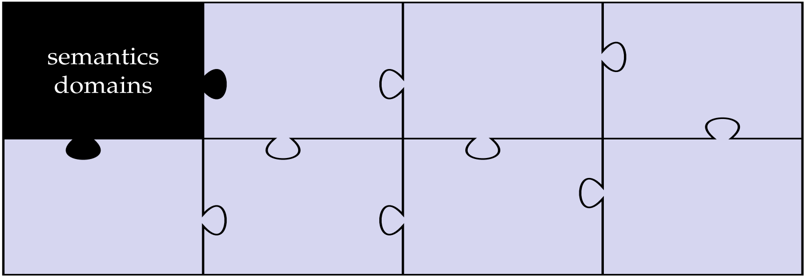
Coq packaging:

```
Record well_formed_program (p:Program) : Set := {
  property1 : ...;
  property2 : ...;
  ...
}.

Definition check_wf (p:Program) :
                    option (well_formed_program P).
```

Proof on wellformed programs:

```
forall (p:Program), well_formed_program p -> ...
```

Verified bytecode verification



Example: JVM states

frame

call stack

$$\langle\langle h, \langle m, pc, l, v :: s\rangle, sf\rangle\rangle$$

heap

method

program point

operand stack

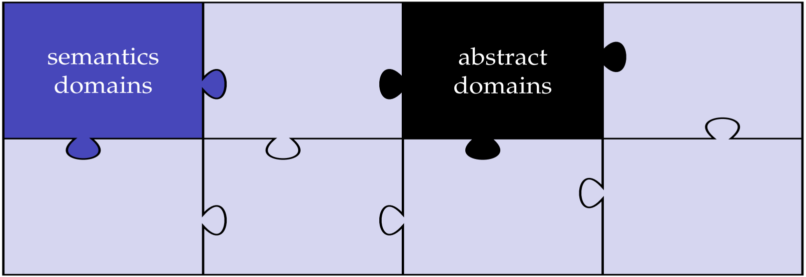
local variables

```
Inductive value : Set :=
      | Int (v:Z) (* Numeric value *)
      | NULL (* reference *)
      | UNDEF (* default value *).

(* Initial (default) value. Must be compatible with the type of the field. *)
    Parameter initValue : Field -> value.


Module Type LOCALVAR.
Parameter t : Type.
   Parameter get : t-> Var -> option value.
   Parameter update : t -> Var -> value -> t.
   Parameter get_update_new : forall l x v, get (update l x v) x = Some v.
   Parameter get_update_old : forall l x y v,
     x<>y -> get (update l x v) y = get l y.
 End LOCALVAR.
 Declare Module LocalVar : LOCALVAR.

Module Type OPERANDSTACK.
   Definition t : Set := list value.
   Definition empty : t := nil.
   Definition push : value -> t -> t := fun v t => cons v t.
   Definition size : t -> nat := fun t  => length t.
   Definition get_nth : t -> nat -> option value := fun s n => nth_error s n.
 End OPERANDSTACK.
 Declare Module OperandStack : OPERANDSTACK.

 (** Transfer fonction between operand stack and local variables **)
 Parameter stack2localvar : OperandStack -> nat -> LocalVar.t.
```

Formalization of JVM states
Heap

```
Module Type HEAP.
  Parameter t : Type.

  Inductive AdressingMode : Set :=
   | StaticField : FieldSignature -> AdressingMode
   | DynamicField : Location -> FieldSignature -> AdressingMode
   | ArrayElement : Location -> Int -> AdressingMode.

  Inductive LocationType : Set :=
   | LocationObject : ClassName -> LocationType
   | LocationArray : Int -> type -> LocationType.
  (** (LocationArray length element_type) *)

  Parameter typeof : t -> Location -> option LocationType.
  (** typeof h loc = None -> no object, no array allocated at location loc *)


  Parameter get : t -> AdressingMode -> option value.
  Parameter update : t -> AdressingMode -> value -> t.

   Parameter new : t -> Program -> LocationType -> option (Location * t).


  Parameter get_update_same : forall h am v, Compat h am ->
                                get (update h am v) am = Some v.

  Parameter get_update_old : forall h am1 am2 v, am1<>am2 ->
                                get (update h am1 v) am2 = get h am2.

  Parameter new_fresh_location :
   forall (h:t) (p:Program) (lt:LocationType) (loc:Location) (h':t),
    new h p lt = Some (loc,h') ->
    typeof h loc = None.

  ...
```

Verified bytecode verification

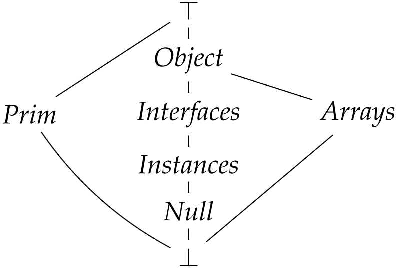


- is partially ordered,
- with a top element $\top$ for errors,
- and a "lub" operator $\sqcup$
- w/o infinite increasing chains

$$x_0 \sqsubset x_1 \sqsubset \cdots \sqsubset \cdots$$

- Inherited from JVM types (extension to finite maps and stacks)

JVM types



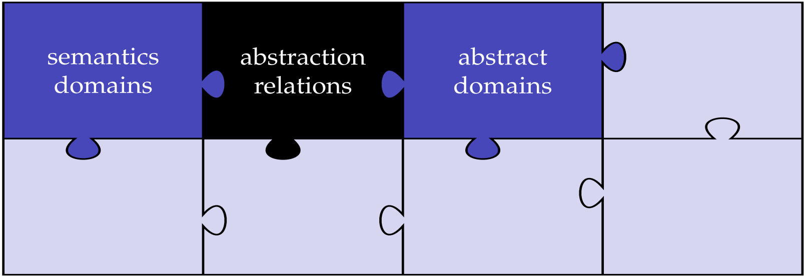

```
Inductive type : Set :=
      | ReferenceType (rt : refType)
      | PrimitiveType (pt: primitiveType)
  with refType :Set :=
      | ArrayType (typ:type)
      | ClassType  (ct:ClassName)
      | InterfaceType (it:InterfaceName)
  with  primitiveType : Set :=
      | BOOLEAN  | BYTE | SHORT | INT.
```

Specific challenges, e.g. interfaces

```
interface I { ... }
interface J { ... }
class C implements I, J { ... }
class D implements I, J { ... }
```

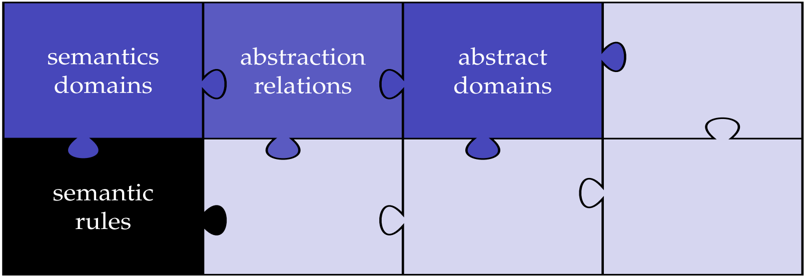
Both I and J are upper bounds for C and D, but they are incomparable.

Verified bytecode verification



- Each type represents a property on concrete values
- This correspondence is formalised by the relation *value* : *type* (that respects subtyping)

Verified bytecode verification



Operational semantics $\rightsquigarrow$ between states

$$\frac{P[(m,pc)] = \mathsf{push}\ c}{\langle\langle h, \langle m, pc, l, s\rangle, sf\rangle\rangle \rightsquigarrow \langle\langle h, \langle m, pc+1, l, c :: s\rangle, sf\rangle\rangle}$$

$$\frac{\begin{array}{c} P[(m,pc)] = \mathsf{invokevirtual}\ m_{id} \\ \begin{array}{rcl} m' & = & \mathrm{methodLookup}(m_{id}, h(loc)) \\ V & = & v_1 :: \cdots :: v_{\mathrm{nbArguments}(m_{id})} \end{array}\end{array}}{\langle\langle h, \langle m, pc, l, loc :: V :: s\rangle, sf\rangle\rangle \rightsquigarrow \langle\langle h, \langle m', 1, V, \varepsilon\rangle, \langle m, pc, l, s\rangle :: sf\rangle\rangle}$$

Formalization of rules

```
| const_step_ok : forall h m pc pc' s l sf t z,
   instructionAt m pc = Some (Const t z) ->
   next m pc = Some pc' ->
   (    (t=BYTE /\ -2^7 <= z < 2^7)
    \/ (t=SHORT /\ -2^15 <= z < 2^15)
    \/ (t=INT /\ -2^31 <= z < 2^31)    ) ->
   step p (St h (Fr m pc s l) sf) (St h (Fr m pc' (Num (I (Int.const z))::s) l) sf)

 | invokevirtual_step_ok : forall h m pc s l sf mid cn M args loc cl bM fnew,
   instructionAt m pc = Some (Invokevirtual (cn,mid)) ->
   lookup p cn mid (pair cl M) ->
   Heap.typeof h loc = Some (Heap.LocationObject cn) ->
   length args = length (METHODSIGNATURE.parameters mid) ->
  METHOD.body M = Some bM ->
   fnew = (Fr M
             (BYTECODEMETHOD.firstAddress bM)
              OperandStack.empty
             (stack2localvar (args++(Ref loc)::s) (1+(length args)))) ->

   step p (St h (Fr m pc (args++(Ref loc)::s) l) sf) (St h fnew ((Fr m pc s l)::sf))
```

Small step semantics

Two kinds of state:

- normal state :

```
(St h (Fr m pc s l) sf)
```

- exception state (not yet caught)

```
(StE h (FrE m pc loc l) sf)
```

The small step semantics is defined with a relation between state

```
step (p:Program) : State -> State -> Prop
```

Small step semantics

Four cases

1. normal $\longrightarrow$ normal
2. normal $\longrightarrow$ exception
3. exception $\longrightarrow$ normal
4. exception $\longrightarrow$ exception

Small step semantics
Four cases

1. normal → normal

```
| putfield_step_ok : forall h m pc pc' s l sf f loc cn v,

  instructionAt m pc = Some (Putfield f) ->
  next m pc = Some pc' ->
  Heap.typeof h loc = Some (Heap.LocationObject cn) ->
  defined_field p cn f ->
  assign_compatible p h v (FIELDSIGNATURE.type f) ->

  step p (St h (Fr m pc (v::(Ref loc)::s) l) sf)
         (St (Heap.update h (Heap.DynamicField loc f) v)
             (Fr m pc' s l) sf)
```

2. normal → exception
3. exception → normal
4. exception → exception

1. normal → normal
2. normal → exception

```
| putfield_step_NullPointerException :
    forall h m pc s l sf f v h' loc',

  instructionAt m pc = Some (Putfield f) ->
  Heap.new h p (Heap.LocationObject
                  (javaLang,NullPointerException))
                                    = Some (loc',h') ->

  step p (St h (Fr m pc (v::Null::s) l) sf)
         (StE h' (FrE m pc loc' l) sf)
```

3. exception → normal
4. exception → exception

Small step semantics

Four cases

1. normal → normal
2. normal → exception
3. exception → normal

```
| exception_caught : forall h m pc loc l sf bm pc',

  METHOD.body m = Some bm ->
  lookup_handlers p
       (BYTECODEMETHOD.exceptionHandlers bm) h pc loc pc' ->

  step p (StE h (FrE m pc loc l) sf)
         (St h (Fr m pc' (Ref loc::nil) l) sf)
```

4. exception → exception

Small step semantics

Four cases

1. normal → normal
2. normal → exception
3. exception → normal
4. exception → exception

```
| exception_uncaught : forall h m pc loc l m' pc' s' l' sf bm,

  METHOD.body m = Some bm ->
  (forall pc'',
     ~ lookup_handlers p
        (BYTECODEMETHOD.exceptionHandlers bm) h pc loc pc'') ->

  step p (StE h (FrE m pc loc l) ((Fr m' pc' s' l')::sf))
         (StE h (FrE m' pc' loc l') sf)
```

Big step semantics

- The small step semantics is not well suited to prove the correctness of moduler verification methods
- Better to reason relative to intermediate semantics with method calls are performed in one-step, or relative to big-step semantics

$$m \vdash \langle h, k, pc, s, l \rangle_{\text{intra}} \Rightarrow^* v$$

- Still necessary to prove correspondence with the small step semantics.

Big step semantics

```
IntraBigStep (P:Program) :
    Method -> IntraNormalState -> ReturnState -> Prop
```

The big step semantics relies on 4 kinds of elementary steps:

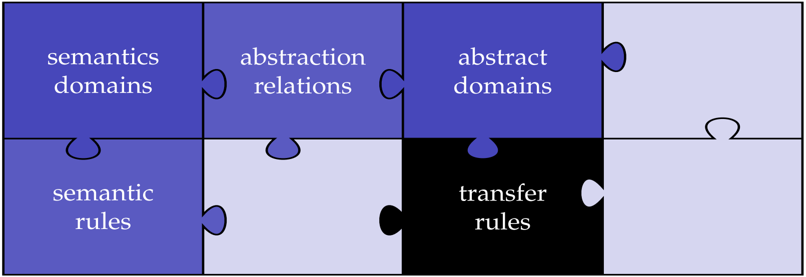
1. normal intra step
2. exception step
3. call step
4. return step

These relations can be combined to obtain different kinds of big step semantics.

Theorem

Big-step semantics and small-step semantics are equivalent (in some precise mathematical sense based on complete executions)

Verified bytecode verification



- the type system is specified by transfer rules

```
tstep (p:Program) : tState -> tState -> Prop.
```

  whose definition is similar to operational semantics
- the definition of typability is a direct application of transfer rules
- a type is a solution of a fixpoint problem $F^\sharp(S) \sqsubseteq (S)$ or equivalently of a constraint system

Sample transfer rules

$$\frac{P[i] = iadd}{i \vdash rt, int :: int :: st \Rightarrow rt, int :: st}$$

$$\frac{P[i] = iconst\ n \quad |st| + 1 \leqslant Mstack}{i \vdash rt, st \Rightarrow rt, int :: st}$$

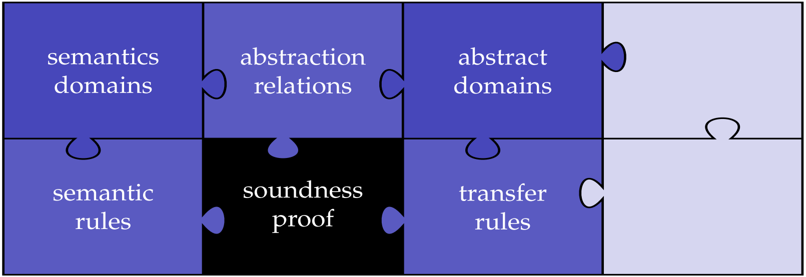
$$\frac{P[i] = aload\ n \quad rt(n) = \tau \quad \tau \prec Object \quad |st| + 1 \leqslant Mstack}{rt, st \Rightarrow rt, \tau :: st}$$

$$\frac{P[i] = astore\ n \quad \tau \prec Object \quad 0 \leqslant n < Mreg}{i \vdash rt, \tau :: st \Rightarrow rt[n \leftarrow \tau], st}$$

$$\frac{P[i] = getfield\ C\ f\ \tau \quad \tau' \prec C}{i \vdash rt, \tau' :: st \Rightarrow rt, \tau :: st}$$

$$\frac{P[i] = putfield\ C\ f\ \tau \quad \tau_1 \prec \tau \quad \tau_2 \prec C}{i \vdash rt, \tau_1 :: \tau_2 :: st \Rightarrow rt, st}$$

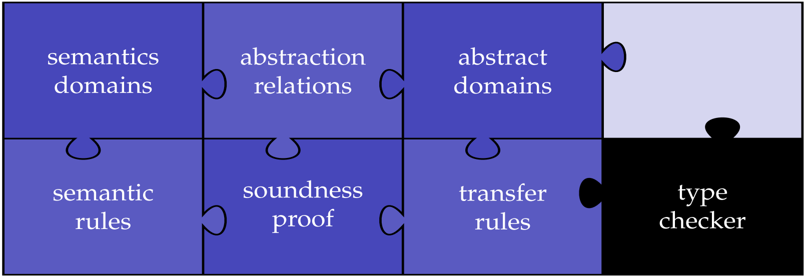
Verified bytecode verification



If $s \rightsquigarrow s'$ and s is type-correct, then s' is type-correct

- easy proof, but tedious: one proof by instruction
- uses intermediate semantics
- exceptions may be handled separately

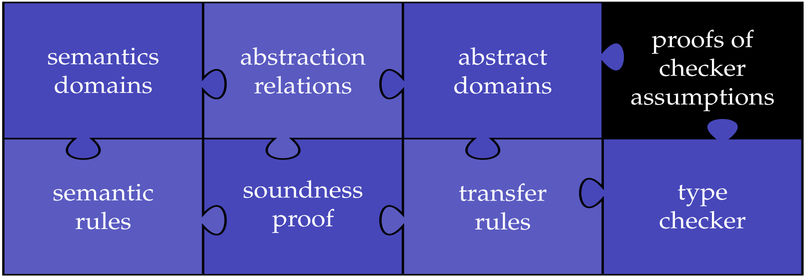
Verified bytecode verification



From declarative definition of typable program to type checker

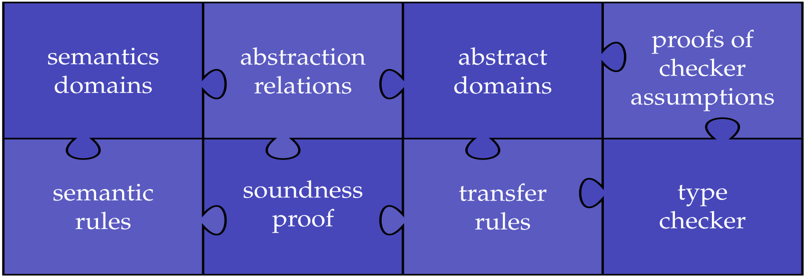
- rely on generic construction
- ... but requires discharging hypotheses!

Verified bytecode verification



- implement functions for inclusion checking
- provide hypotheses that guarantee termination (for bcv, not lbcv)

Verified bytecode verification



Final results

$$\left.\begin{array}{c}\mathsf{check}\ P = ok \\ \mathsf{s_{init}} \Downarrow \mathsf{s_{final}} \\ \mathsf{s_{init}}\ \text{type}-\text{correct}\end{array}\right\} \Rightarrow \mathsf{s_{final}}\ \text{type}-\text{correct}$$

- progress
- commutation defensive and offensive machine

Beyond bytecode verification

- Types are properties:
  - being an integer
  - being a boolean
- More precise types:
  - parity
  - interval
  - etc.

- Properties organized as a lattice of abstract elements.
- Transfer rules capture abstract behavior of functions

Examples

Parity

- Abstract properties

$$\text{odd} \qquad \text{even}$$

- Least upper bound

$$\text{odd} \sqcup \text{even} = \top$$

- Abstract semantics of addition

$$\begin{array}{rcl} \text{even} + \text{even} & = & \text{even} \\ \text{odd} + \text{odd} & = & \text{even} \\ \text{even} + \text{odd} & = & \text{odd} \\ x + \top & = & \top \\ x + \bot & = & \bot \\ \ldots & & \ldots \end{array}$$

Intervals

- Abstract properties

$$[i, j]$$

where $i, j \in \text{int} \sqcup \{+\infty, -\infty\}$

- Least upper bound

$$[i, j] \sqcup [i', j'] = [i'', j'']$$

where

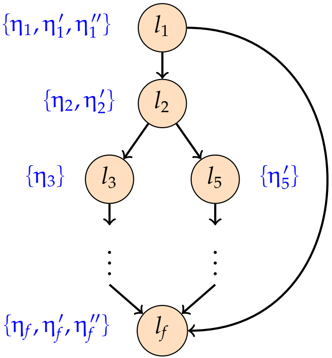
$$\begin{array}{rcl} i'' & = & min(i, i') \\ j'' & = & max(j, j') \end{array}$$

- Abstract semantics of addition

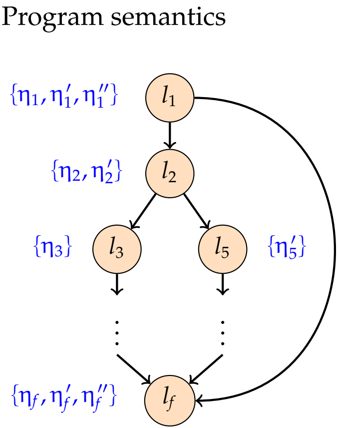
$$[i, j] + [i', j'] = [i + i', j + j']$$

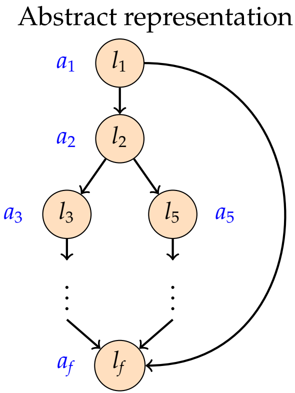
Concrete vs abstract semantics

Program semantics

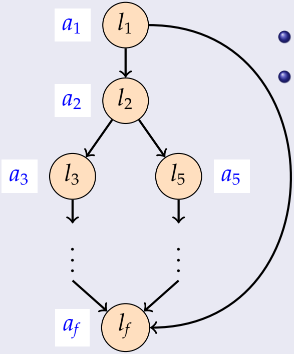


Concrete vs abstract semantics





Solution



- $\mathbf{D}^\sharp = \langle D^\sharp, \sqsubseteq, \sqcap, \ldots \rangle$,
- $T_{\langle l_i, l_j \rangle} : D^\sharp \to D^\sharp$ a monotonic transfer function (for any edge $\langle l_i, l_j \rangle$)

$\{a_1, a_2, \ldots, a_f\}$ a solution of $(\mathbf{D}, T)$ if:

$$T_{\langle l_1, l_2 \rangle}(a_1) \sqsubseteq a_2$$
$$T_{\langle l_2, l_5 \rangle}(a_2) \sqsubseteq a_5$$
$$T_{\langle l_1, l_f \rangle}(a_1) \sqsubseteq a_f$$
$$\ldots$$

Soundness w.r.t. program semantics (D, T): for all $d : D$ and edge e

$$\alpha(T_e\, d) \sqsubseteq T_e^\sharp\, (\alpha\, d)$$

Partial solution

- A partial annotation map is a partial mapping $S : \mathcal{P} \rightharpoonup \mathcal{A}$
  - partial annotations generalize stackmaps
- May be extended to $\hat{S} : \mathcal{P} \to \mathcal{A}$

$$\hat{S}(l') = \bigcup_{\langle l,l' \rangle \in \mathcal{E}} T_{\langle l,l' \rangle}(\hat{S}(l))$$

  - provided the domain of S is sufficiently large

However checking $\sqsubseteq$ may be...

- Expensive
- Undecidable

Certified solution

$\langle\{a_1 \ldots a_n\}, c\rangle$ is a certified solution if for any edge $\langle i, j\rangle$

$$c(i,j) \in \mathcal{C}(\vdash T_{\langle i,j\rangle}(a_i) \sqsubseteq a_j)$$

- Every certified solution is a solution
- A solution can be certified by exhibiting certificates:

If $\{a_1 \ldots a_n\}$ is a solution of $(D^\sharp, T^\sharp)$, and cons s.t. for any edge $\langle i, j\rangle$

$$\mathsf{cons}_{\langle i,j\rangle} \in \mathcal{C}(\vdash T_{\langle i,j\rangle}(\gamma(a)) \sqsubseteq \gamma(T^\sharp_{\langle i,j\rangle}(a)))$$

then $(\{\gamma(a_1) \ldots \gamma(a_n)\}, c)$ is a certified solution of (D, T) [for some c].

Abstraction-Carrying Code

- Powerful generalization of lightweight bytecode verification
- Programs come equipped with a partial solution
- One pass verification (decidable assuming $\sqsubseteq$ is decidable)
- May embed a notion of certificate

Verified abstraction carrying code

It is possible to generalize verified bytecode verification to verified abstraction carrying code

- Resource control
- Array-out-of-bound exceptions
- Non-interference

- Generic lattice library
- General lemmas about well-founded orders

Example of certified analyzer: memory consumption

- The goal of the type system is to provide an upper bound on the number of dynamically created objects.
- Jugdments are of the form $\vdash P : n$ to indicate that P creates at most n objects.

Transfer rules

$$\frac{P[i] = \mathsf{new}, \mathsf{newarray}}{i \vdash n \Rightarrow n+1} \qquad \frac{P[i] \neq \mathsf{new}, \mathsf{newarray}}{i \vdash n \Rightarrow n}$$

Bounded programs

Typing rule for source level programs:

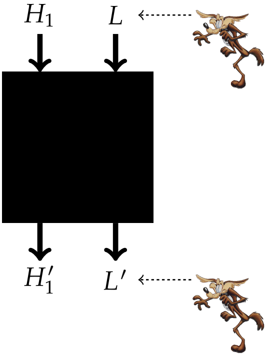
$$\frac{c : 0}{\textsf{while } b \textsf{ do } c : 0}$$

One can enforce a similar constraint for bytecode using widening

- A program is bounded iff for every i s.t. $P[i] = \textsf{new}, \textsf{newarray}$, i is not in a loop, i.e. $i \not\mapsto^{+} i$
- Assume P is safe. Then P is bounded iff there exists n s.t. $\vdash P : n$.

Non-interference

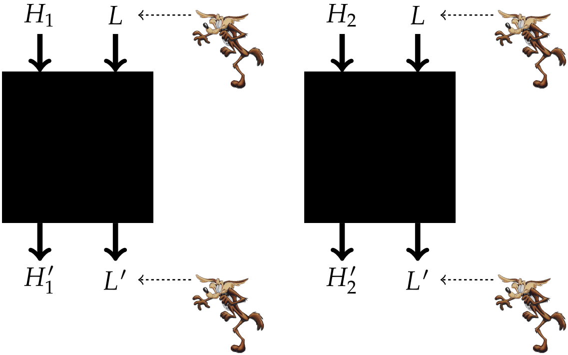
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidential   Low = public

Non-interference

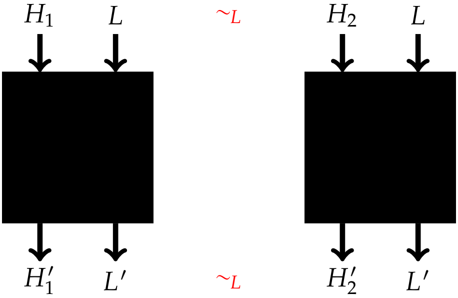
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidential   Low = public

Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



$$\forall s_1, s_2,\ s_1 \sim_L s_2 \wedge P, s_1 \Downarrow s_1' \wedge P, s_2 \Downarrow s_2' \implies s_1' \sim_L s_2'$$

High = confidential   Low = public

Simple bytecode language SBC

A program is an array of instructions:

$instr$	$::=$	prim op	primitive operation
	$\mid$	push v	push value on top of stack
	$\mid$	load x	load value of x on stack
	$\mid$	store x	store top of stack in x
	$\mid$	if j	conditional jump
	$\mid$	goto j	unconditional jump
	$\mid$	return	return

where:

- $j \in \mathcal{P}$ is a program point
- $v \in \mathcal{V}$ is a value
- $x \in \mathcal{X}$ is a variable

Semantics

- States are of the form $\langle\langle i, \rho, s \rangle\rangle$ where:
  - $i : \mathcal{P}$ is the program counter
  - $\rho : \mathcal{X} \to \mathcal{V}$ maps variables to values
  - $s : \mathcal{V}^\star$ is the operand stack
- Operational semantics is given by rules are of the form

$$\frac{P[i] = ins \qquad constraints}{s \leadsto s'}$$

- Evaluation semantics: $P, \mu \Downarrow \nu, v$ iff $\langle\langle 1, \mu, \epsilon \rangle\rangle \leadsto^\star \langle\langle \nu, v \rangle\rangle$, where $\leadsto^\star$ is the reflexive transitive closure of $\leadsto$

Semantics: rules

$$\frac{P[i] = \textsf{prim}\ op \qquad n_1\ \underline{op}\ n_2 = n}{\langle\langle i, \rho, n_1 :: n_2 :: s \rangle\rangle \rightsquigarrow \langle\langle i+1, \rho, n :: s \rangle\rangle}$$

$$\frac{P[i] = \textsf{push}\ n}{\langle\langle i, \rho, s \rangle\rangle \rightsquigarrow \langle\langle i+1, \rho, n :: s \rangle\rangle}$$

$$\frac{P[i] = \textsf{load}\ x}{\langle\langle i, \rho, s \rangle\rangle \rightsquigarrow \langle\langle i+1, \rho, \rho(x) :: s \rangle\rangle}$$

$$\frac{P[i] = \textsf{store}\ x}{\langle\langle i, \rho, v :: s \rangle\rangle \rightsquigarrow \langle\langle i+1, \rho(x := v), s \rangle\rangle}$$

$$\frac{P[i] = \textsf{if}\ j}{\langle\langle i, \rho, \mathit{false} :: s \rangle\rangle \rightsquigarrow \langle\langle j, \rho, s \rangle\rangle}$$

$$\frac{P[i] = \textsf{if}\ j}{\langle\langle i, \rho, \mathit{true} :: s \rangle\rangle \rightsquigarrow \langle\langle i+1, \rho, s \rangle\rangle}$$

$$\frac{P[i] = \textsf{goto}\ j}{\langle\langle i, \rho, s \rangle\rangle \rightsquigarrow \langle\langle j, \rho, s \rangle\rangle}$$

$$\frac{P[i] = \textsf{return}}{\langle\langle i, \rho, v :: s \rangle\rangle \rightsquigarrow \langle\langle \rho, v \rangle\rangle}$$

Examples of insecure programs

Direct flow

load y_H
store x_L
return

Indirect flow

load y_H
if 5
push 0
store x_L
return

Flow via return

load y_H
if 5
push 1
return
push 0
return

Flow via operand stack

push 0
push 1
load y_H
if 6
swap
store x_L
return 0

Policy

- A lattice of security levels $\mathcal{S} = \{H, L\}$ with $L \leqslant H$
- Each program is given a security signature: $\Gamma : \mathcal{X} \to \mathcal{S}$ and k_{ret}.
- Γ determines an equivalence relation $\sim_L$ on memories: $\rho \sim_L \rho'$ iff

$$\forall x \in \mathcal{X}. \Gamma(x) \leqslant L \Rightarrow \rho(x) = \rho'(x)$$

- Program P is *non-interfering* w.r.t. signature Γ, k_{ret} iff for every $\mu, \mu', \nu, \nu', v, v'$,

$$\left.\begin{array}{c} P, \mu \Downarrow \nu, v \\ P, \mu' \Downarrow \nu', v' \\ \mu \sim_L \mu' \end{array}\right\} \Rightarrow \nu \sim_L \nu' \wedge (k_{\mathrm{ret}} \leqslant L \Rightarrow v = v')$$

Type system

- Transfer rules of the form

$$\frac{P[i] = ins \qquad constraints}{i \vdash st \Rightarrow st'} \qquad \frac{P[i] = ins \qquad constraints}{i \vdash st \Rightarrow}$$

  where $st, st' \in \mathcal{S}^\star$.

- Types assign stack of security levels to program points

$$S : \mathcal{P} \to \mathcal{S}^\star$$

- $S \vdash P$ iff $S_1 = \epsilon$ and for all $i, j \in \mathcal{P}$
  - $i \mapsto j \Rightarrow \exists st'. \, i \vdash S_i \Rightarrow st' \land st' \leqslant S_j$;
  - $i \mapsto \Rightarrow \; i \vdash S_i \Rightarrow$

The transfer rules and typability relation are implicitly parametrized by a signature Γ, k_{ret} and additional information (next slide)

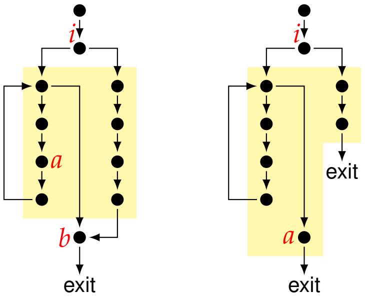
A program point j is in a *control dependence region* of a branching point i if

- j is reachable from i,
- there is a path from i to a return point which does not contain j

CDR can be computed using post-dominators of branching points.

Example :

- a must belong to $region(i)$
- b does not necessary belong to $region(i)$



CDR usage : tracking implicit flows

In a typical type system for a structured language:

$$\frac{\vdash exp : k \qquad [k_1] \vdash c_1 \qquad [k_2] \vdash c_2 \qquad k \leqslant k_1 \qquad k \leqslant k_2}{[k] \vdash \text{if } exp \text{ then } c_1 \text{ else } c_2}$$

In our context

- *se*: a security environment that attaches a security level to each program point
- for each branching point i, we constrain $se(j)$ for all $j \in region(i)$

$$\frac{P[i] = \mathsf{if}\ i' \qquad \forall j \in region(i),\ k \leqslant se(j)}{i \vdash k :: st \Rightarrow \cdots}$$

CDR soundness is ensured by local conditions (instead of path properties) using $region \in \mathcal{P} \rightarrow \wp(\mathcal{P})$ and $jun \in \mathcal{P} \rightharpoonup \mathcal{P}$.

SOAP1: for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in region(i)$ or $k = jun(i)$;

SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;

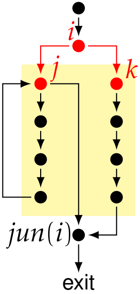
SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.

CDR soundness is ensured by local conditions (instead of path properties) using *region* $\in \mathcal{P} \rightarrow \wp(\mathcal{P})$ and *jun* $\in \mathcal{P} \rightharpoonup \mathcal{P}$.

SOAP1: for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in region(i)$ or $k = jun(i)$;

SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;

SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.

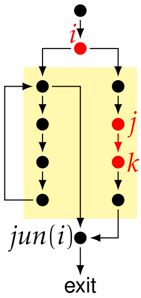


CDR soundness is ensured by local conditions (instead of path properties) using $region \in \mathcal{P} \to \wp(\mathcal{P})$ and $jun \in \mathcal{P} \rightharpoonup \mathcal{P}$.

SOAP1: for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in region(i)$ or $k = jun(i)$;

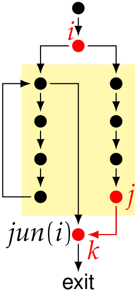
SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;

SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.



CDR soundness is ensured by local conditions (instead of path properties) using *region* $\in \mathcal{P} \to \wp(\mathcal{P})$ and *jun* $\in \mathcal{P} \rightharpoonup \mathcal{P}$.

- SOAP1: for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in region(i)$ or $k = jun(i)$;
- SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;
- SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.



CDR soundness

SOAP (Safe Over Approximation Properties)

CDR soundness is ensured by local conditions (instead of path properties) using *region* $\in \mathcal{P} \rightarrow \wp(\mathcal{P})$ and *jun* $\in \mathcal{P} \rightharpoonup \mathcal{P}$.

SOAP1: for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in region(i)$ or $k = jun(i)$;

SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;

SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.