Language-based methods for software security

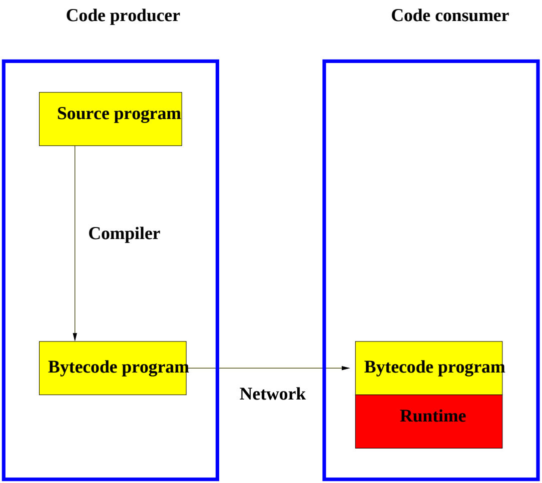
Gilles Barthe

IMDEA Software, Madrid, Spain

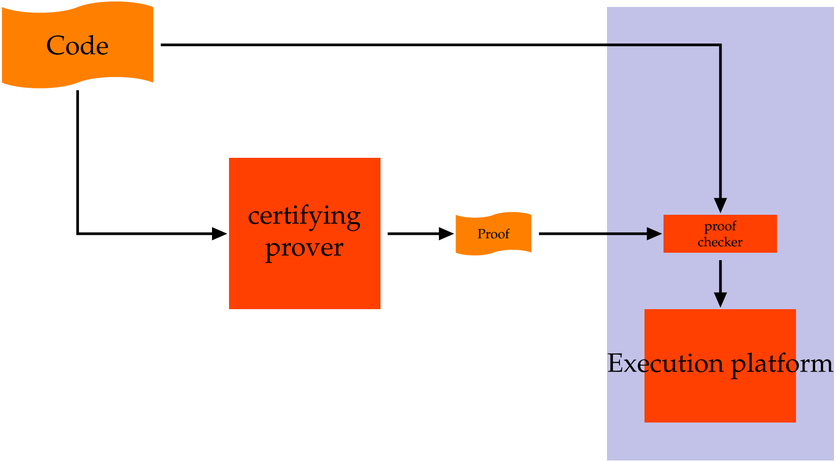
Motivation

- 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

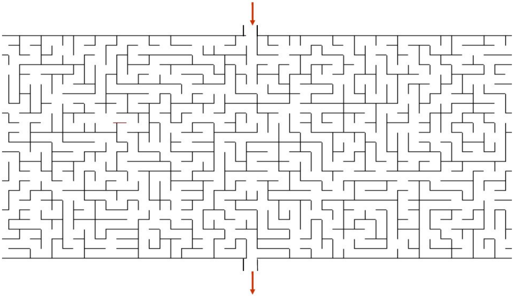


Certificates

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

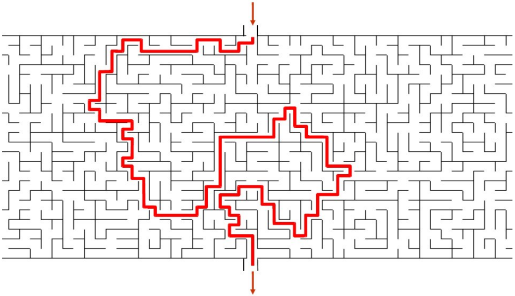
Certificates

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



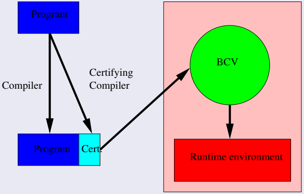
Certificates

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



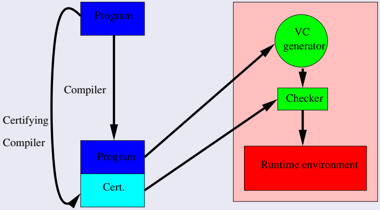
Flavors of Proof Carrying Code

Type-based PCC



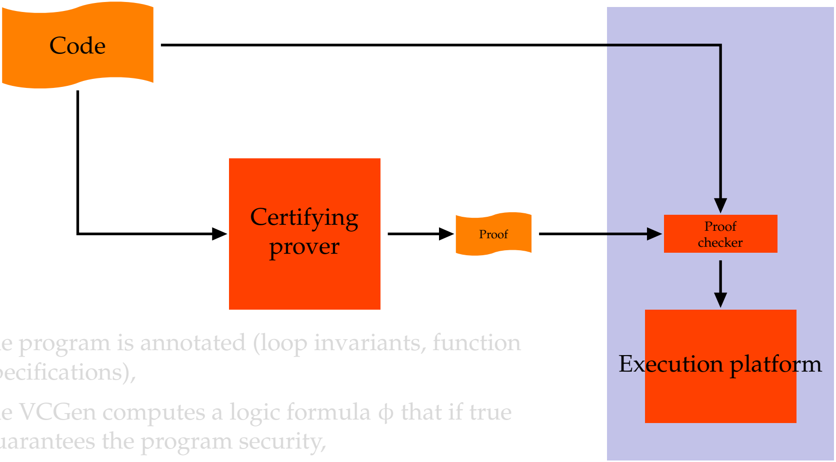
- Widely deployed in KVM
- Application to JVM typing
- On-device checking possible

Logic-based PCC



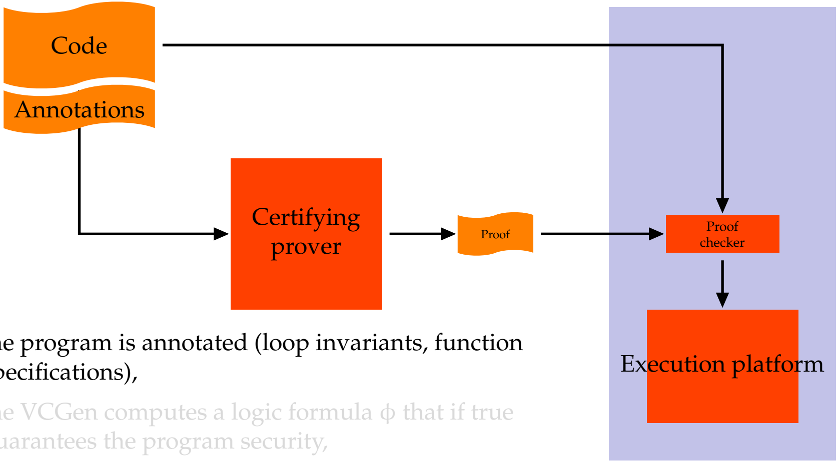
- Original scenario
- Application to type safety and memory safety

Proof carrying code: standard framework



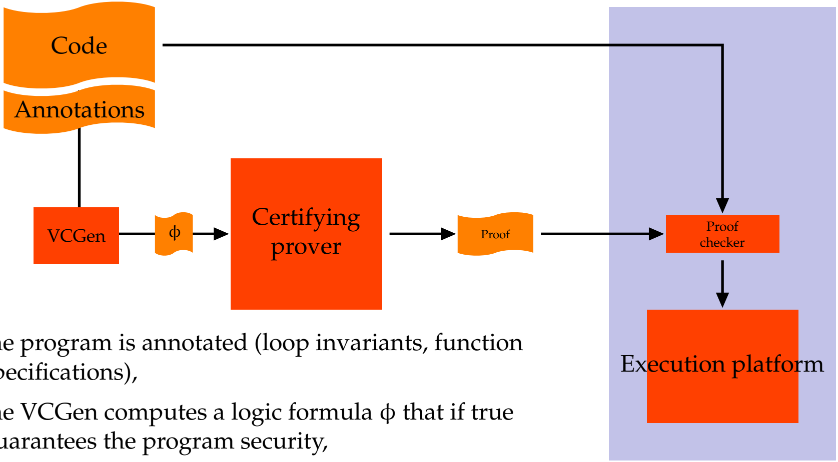
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π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

Proof carrying code: standard framework



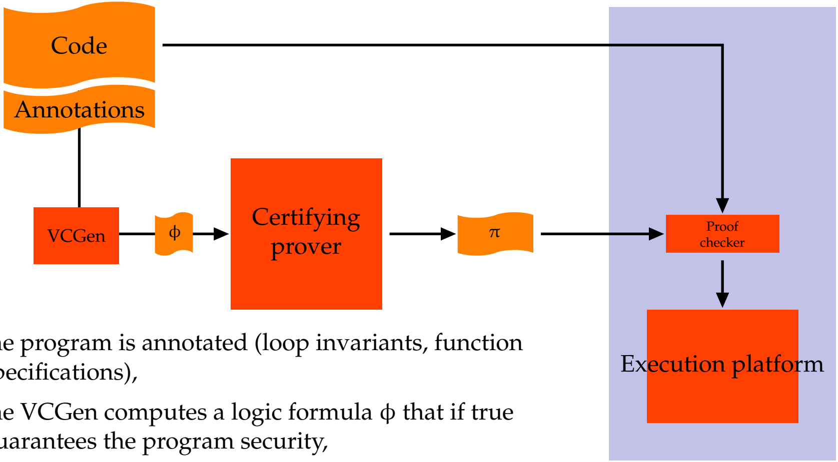
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π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

Proof carrying code: standard framework



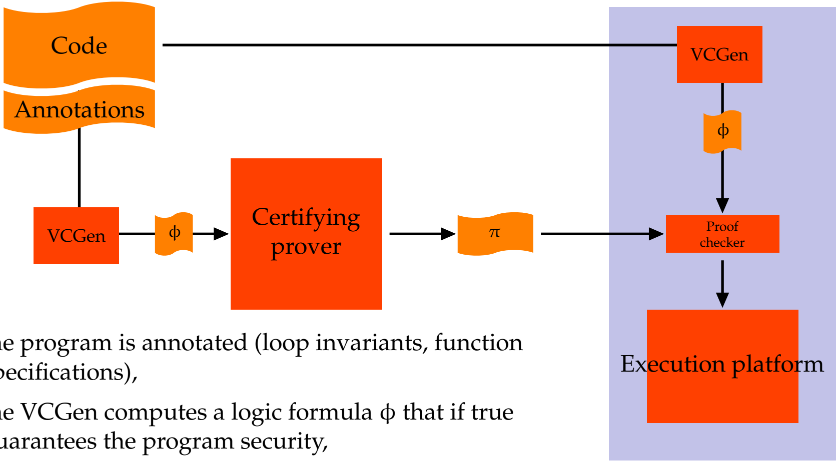
- the program is annotated (loop invariants, function specifications),
- the VCGen computes a logic formula ϕ that if true guarantees the program security,
- the certifying prover computes a *proof object* π which establishes the validity of ϕ,
- the consumer rebuilds the formula ϕ and checks that π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

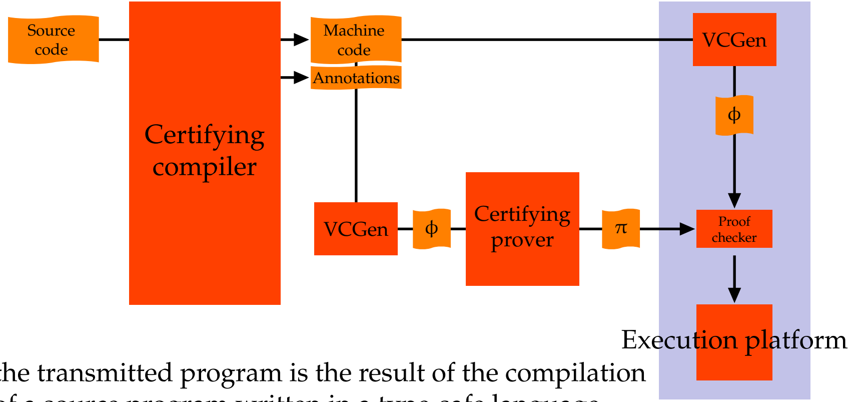
Certifying prover

- 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



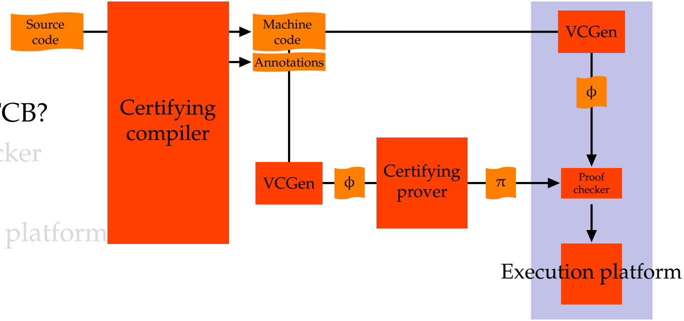
- the transmitted program is the result of the compilation of a source program written in a type-safe language
- the role of the certifying compiler is
  - to check type-safety of the source program
  - to generate corresponding annotations in the machine code to help the VCGen

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.

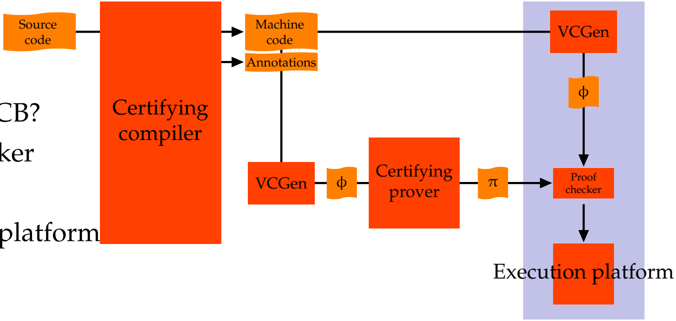
What is the PCC TCB?

- 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?

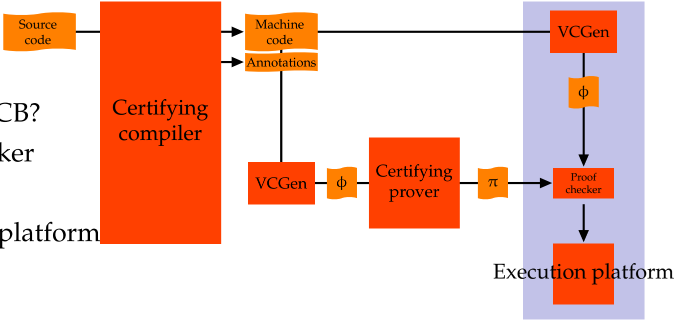
- the proof checker
- the VCGen
- the Execution platform

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



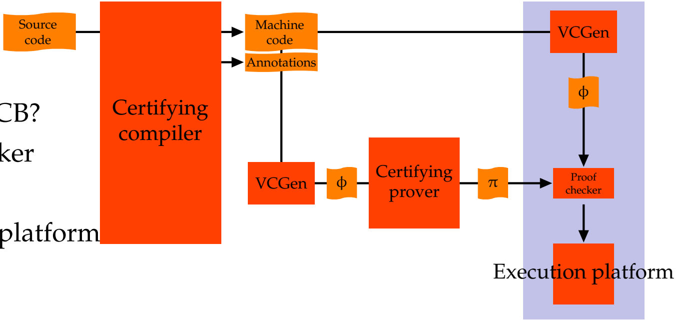
You don't need to trust ...

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?

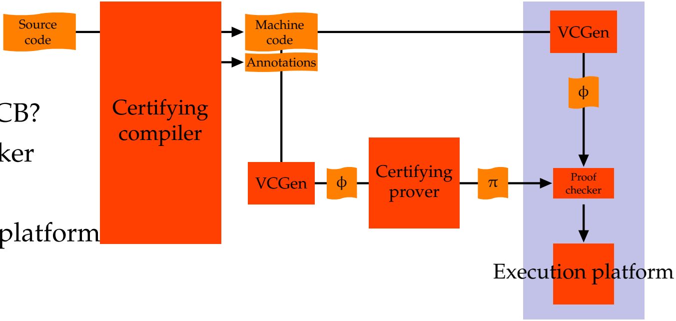
- the proof checker
- the VCGen
- the Execution platform



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

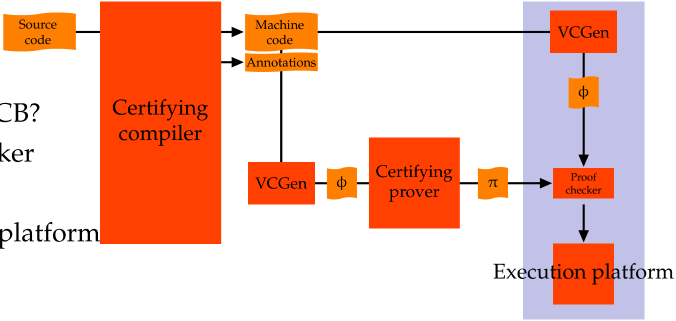
You don't need to trust the compiler, the annotations ...

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



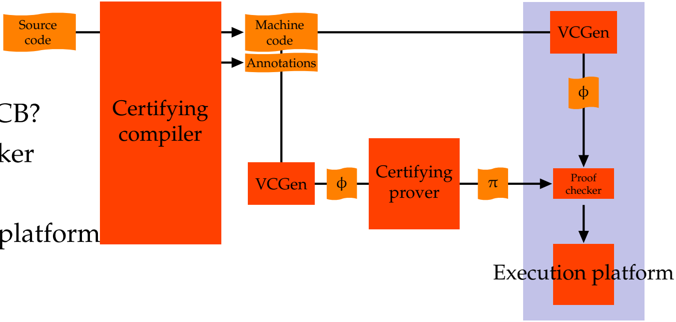
You don't need to trust the compiler, the annotations, the prover ...

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

Simpler checkers?

Proof

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

$$= \quad \wr\text{def. (110) of } \dot{\ddot{\alpha}}\llbracket P\rrbracket \wr$$

$$\ddot{\alpha}\llbracket P\rrbracket \circ \mathrm{Post}\llbracket \mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}\rrbracket \circ \ddot{\gamma}\llbracket P\rrbracket$$

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

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

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

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

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

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

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

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

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

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

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

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

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

Simpler checkers?

Proof

$$
\begin{aligned}
& \dot{\ddot{\alpha}}\llbracket P\rrbracket(\mathrm{Post}\llbracket \mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}\rrbracket) \\
= \quad & \wr \text{def. (110) of } \dot{\ddot{\alpha}}\llbracket P\rrbracket \int \\
& \ddot{\alpha}\llbracket P\rrbracket \circ \mathrm{Post}\llbracket \mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{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 \mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{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 } \mathtt{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\ ?\ J_{\mathrm{at}_P\llbracket S_t\rrbracket} \mathbin{\dot\sqcup} \mathrm{Abexp}\llbracket B\rrbracket(J_\ell)\ ¿\ 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'\ ?\ J^{t''}_{\ell'} \mathbin{\dot\sqcup} J^{t''}_{\mathrm{after}_P\llbracket S_t\rrbracket}\ ¿\ 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\ ?\ J_{\mathrm{at}_P\llbracket S_f\rrbracket} \mathbin{\dot\sqcup} \mathrm{Abexp}\llbracket T(\neg B)\rrbracket(J_\ell)\ ¿\ 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'\ ?\ J^{f''}_{\ell'} \mathbin{\dot\sqcup} J^{f''}_{\mathrm{after}_P\llbracket S_f\rrbracket}\ ¿\ 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\ ?\ J_{\mathrm{at}_P\llbracket S_t\rrbracket} \mathbin{\dot\sqcup} \mathrm{Abexp}\llbracket B\rrbracket(J_\ell)\ ¿\ 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\ ?\ J_{\mathrm{at}_P\llbracket S_f\rrbracket} \mathbin{\dot\sqcup} \mathrm{Abexp}\llbracket T(\neg B)\rrbracket(J_\ell)\ ¿\ 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'\ ?\ 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}\ ¿\ 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'} \\
& \ = 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

$$
\begin{array}{ll}
 & \dot{\ddot{\alpha}}\llbracket P\rrbracket(\mathrm{Post}\llbracket \mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}\rrbracket) \\
= & \quad \wr \text{def. (110) of } \dot{\ddot{\alpha}}\llbracket P\rrbracket \wr \\
 & \ddot{\alpha}\llbracket P\rrbracket \circ \mathrm{Post}\llbracket \mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}\rrbracket \circ \ddot{\gamma}\llbracket P\rrbracket \\
= & \quad \wr \text{def. (103) of Post} \wr \\
 & \ddot{\alpha}\llbracket P\rrbracket \circ \mathrm{post}[\tau^\star\llbracket \mathbf{if}\ B\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f\ \mathbf{fi}\rrbracket] \circ \ddot{\gamma}\llbracket P\rrbracket \\
= & \quad \wr \text{big step operational semantics (93)} \wr \\
 & \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} \wr \\
 & \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)] \ \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} \wr \\
 & (\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) \ \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{le} \\
 & \lambda J \cdot \text{let} \\
 & \qquad \text{le} \\
 & \qquad \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \cdot (l = \ell' \ ?\ J_{\ell'} \ \dot{\sqcup}\ J_{\mathrm{after}_P\llbracket S_t\rrbracket} \ ¿\ J_l) \\
 & \quad \ddot{\sqcup} \\
 & \quad \text{let } J^{f'} = \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \cdot (l = \mathrm{at}_P\llbracket S_f\rrbracket \ ?\ J_{\mathrm{at}_P\llbracket S_f\rrbracket} \ \dot{\sqcup}\ \mathrm{Abexp}\llbracket T(\neg B)\rrbracket(J_\ell) \ ¿\ J_l) \text{ in} \\
 & \qquad \text{let } J^{f''} = \mathrm{APost}\llbracket S_f\rrbracket(J^{f'}) \text{ in} \\
 & \qquad \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \cdot (l = \ell' \ ?\ J^{f''}_{\ell'} \ \dot{\sqcup}\ J^{f''}_{\mathrm{after}_P\llbracket S_f\rrbracket} \ ¿\ J^{f''}_l) \\
= & \quad \wr \text{by grouping similar terms} \wr \\
 & \lambda J \cdot \text{let } J^{t'} = \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \cdot (l = \mathrm{at}_P\llbracket S_t\rrbracket \ ?\ J_{\mathrm{at}_P\llbracket S_t\rrbracket} \ \dot{\sqcup}\ \mathrm{Abexp}\llbracket B\rrbracket(J_\ell) \ ¿\ J_l) \\
 & \quad \text{and } J^{f'} = \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \cdot (l = \mathrm{at}_P\llbracket S_f\rrbracket \ ?\ J_{\mathrm{at}_P\llbracket S_f\rrbracket} \ \dot{\sqcup}\ \mathrm{Abexp}\llbracket T(\neg B)\rrbracket(J_\ell) \ ¿\ J_l) \text{ in} \\
 & \qquad \text{let } J^{t''} = \mathrm{APost}\llbracket S_t\rrbracket(J^{t'}) \\
 & \qquad \text{and } J^{f''} = \mathrm{APost}\llbracket S_f\rrbracket(J^{f'}) \text{ in} \\
 & \qquad \lambda l \in \mathrm{in}_P\llbracket P\rrbracket \cdot (l = \ell' \ ?\ J^{t''}_{\ell'} \ \dot{\sqcup}\ J^{t''}_{\mathrm{after}_P\llbracket S_t\rrbracket} \ \dot{\sqcup}\ J^{f''}_{\ell'} \ \dot{\sqcup}\ J^{f''}_{\mathrm{after}_P\llbracket S_f\rrbracket} \ ¿\ J^{t''}_l \ \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} \wr
\end{array}
$$

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

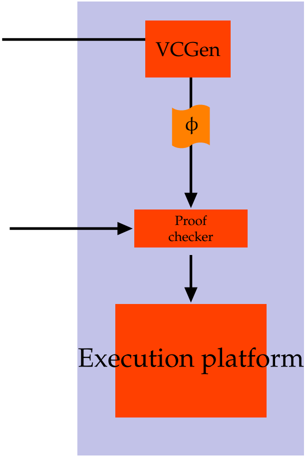
Use computations instead of deductions!

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

If $f\ a$ reduces to true, then $C\ a\ (\texttt{refl\_eq}\ \text{true})$ 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

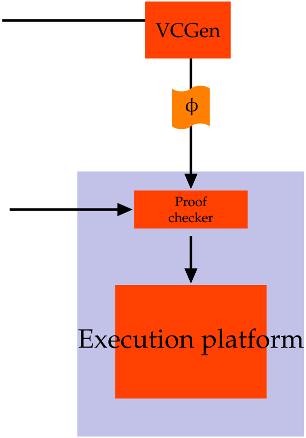


1. In standard PCC
2. If the VCGen is proved correct

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

   (same as FPCC)

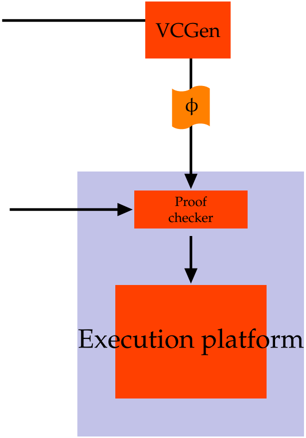
TCB of certified PCC



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)

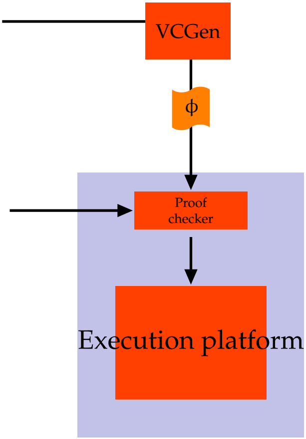
TCB of certified PCC



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)

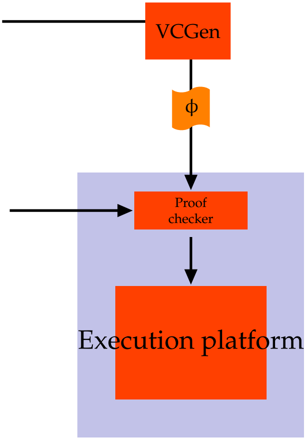
TCB of certified PCC



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

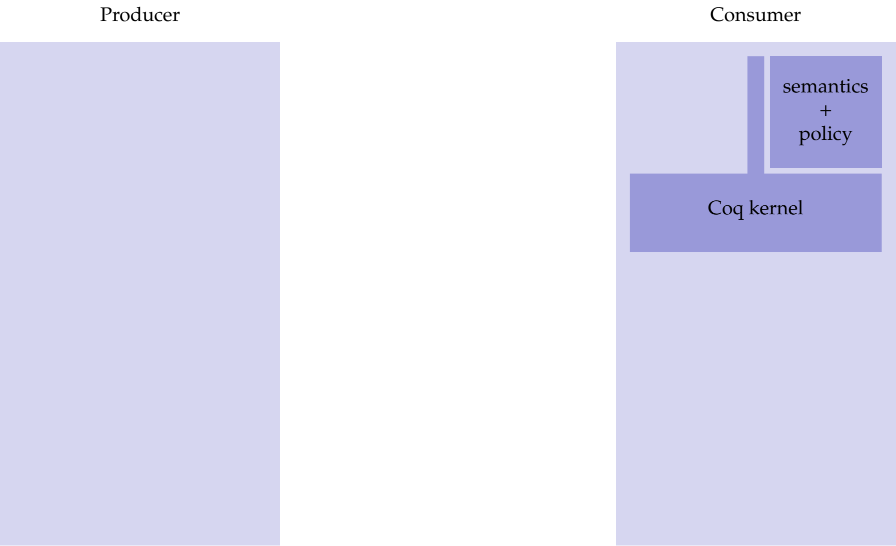
   (same as FPCC)

Using executable checkers

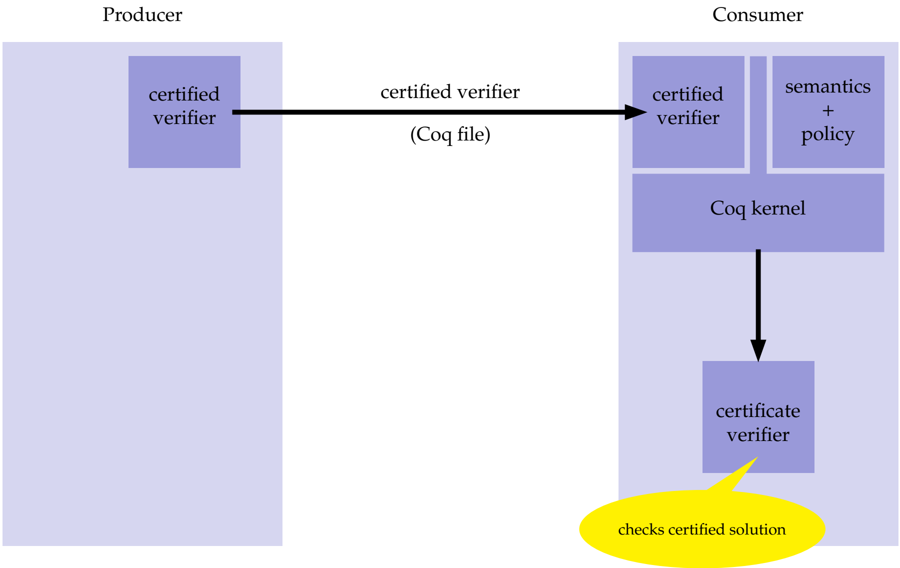
Producer

Consumer

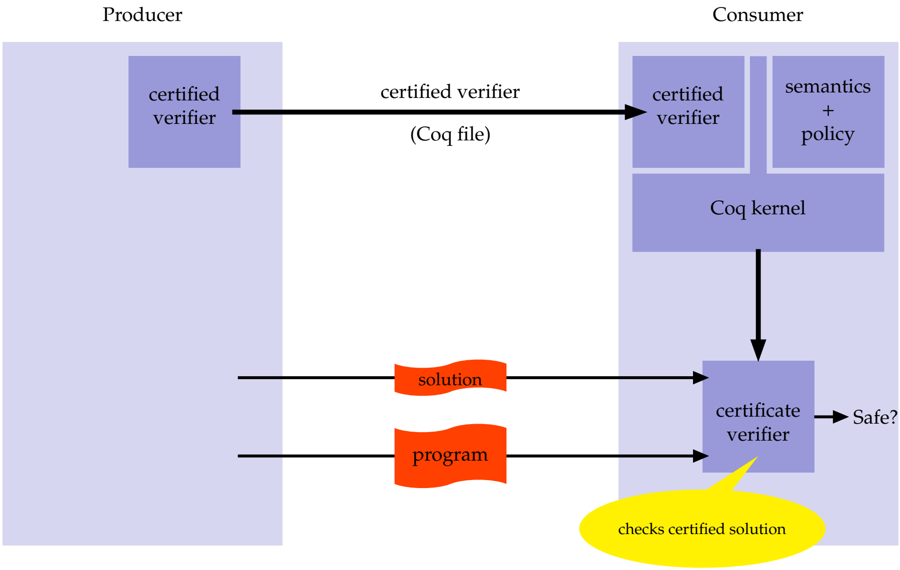
Using executable checkers



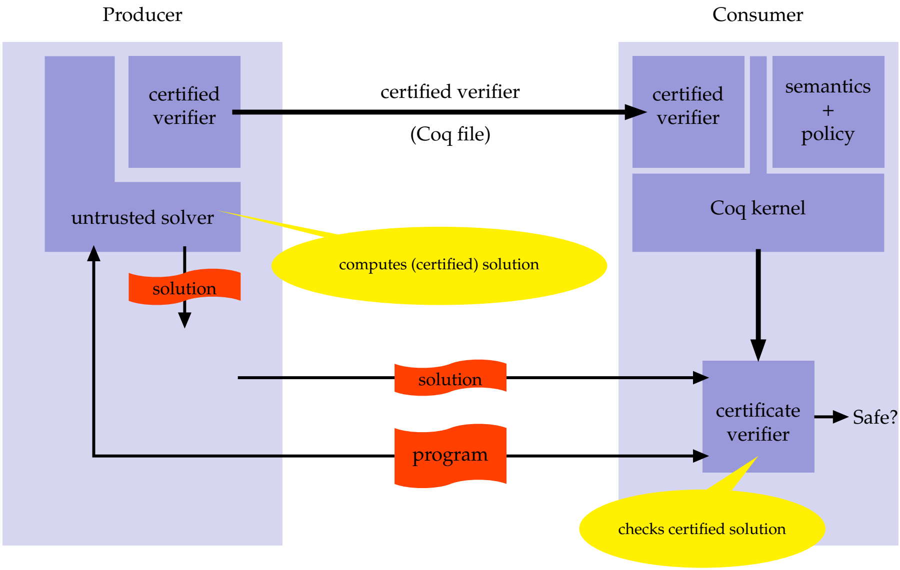
Using executable checkers



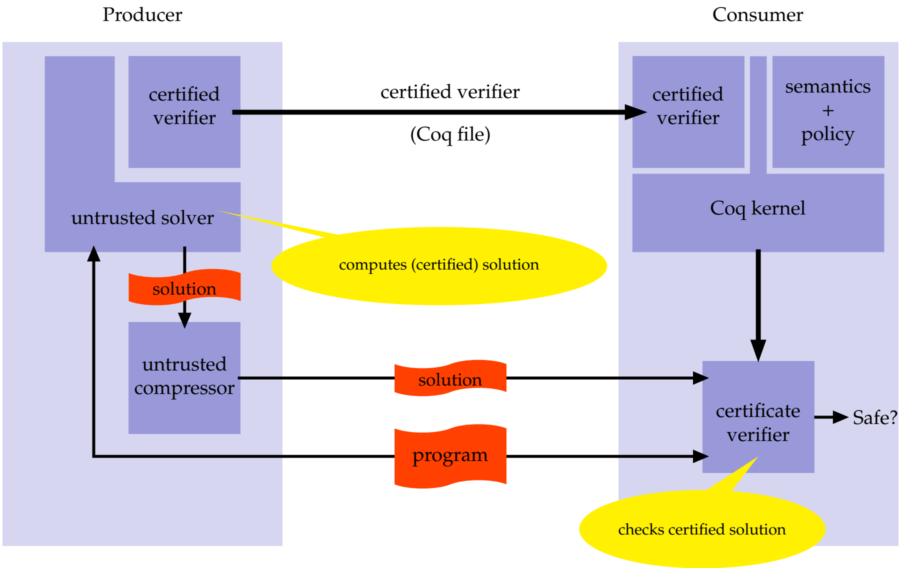
Using executable checkers



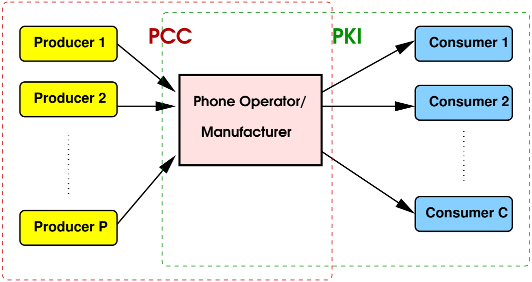
Using executable checkers



Using executable checkers

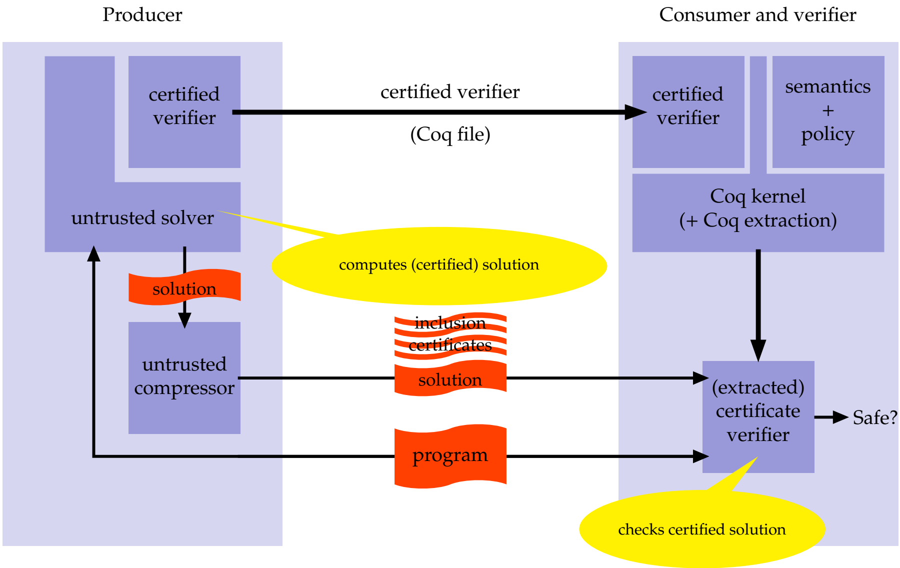


Application scenario: PCC with trusted intermediaries



- 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

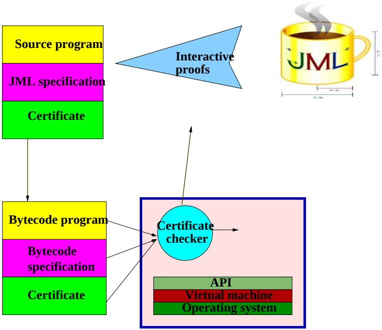
Present two instances of **certified Proof Carrying code** and provide methods to generate certificates from **source code verification**

- 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**

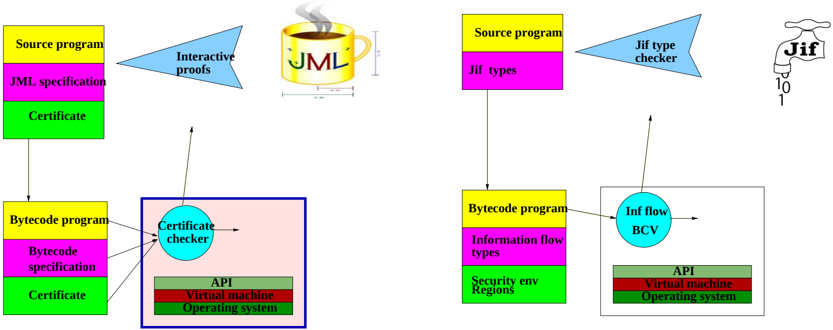
- Type system for information flow based confidentiality policies
- Verification condition generator for logical specifications



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 $\mathsf{mem} = \mathsf{loc} \to \mathsf{Nat}$

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

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

- 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 $\mathsf{mem} = \mathsf{loc} \to \mathsf{Nat}$

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

Evaluation rules

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

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

$\vdash$ *pop* : $(rt, \alpha :: s) \Rightarrow (rt, s)$ $\nvdash$ *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
- $s \leadsto s'$ 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. $\leadsto_{\mathrm{def}} \subseteq \mathsf{tstate} \times (\mathsf{tstate} + \{\mathbf{TypeError}\})$
- If P is type-safe w.r.t. S, then executions of $\leadsto$ and $\leadsto_{\mathrm{def}}$ 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

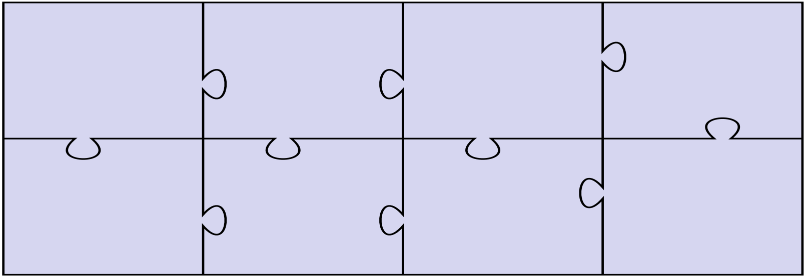
Lightweight bytecode verification

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,
- ...

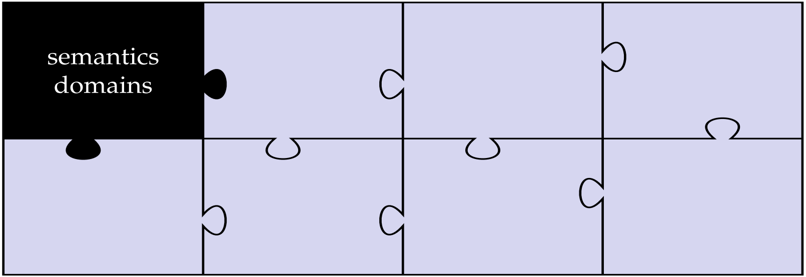
Coq packaging:

```
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

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

heap, method, program point, local variables, operand stack, frame, call stack

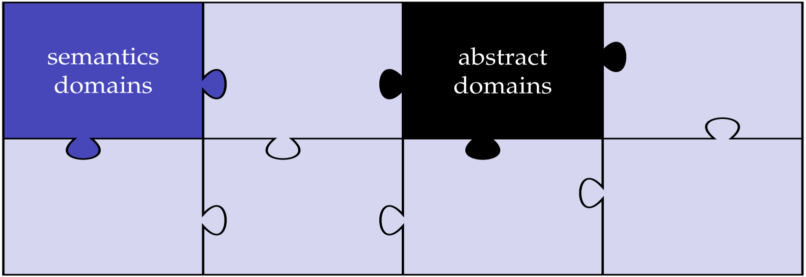
Formalization of JVM states
Values, local variables and operand stack

```
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

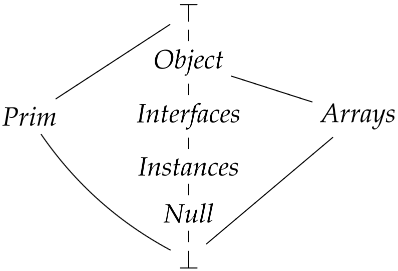


- is partially ordered,
- with a top element $\top$ for errors,
- and a "lub" operator $\sqcup$
- w/o infinite increasing chains

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

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

JVM types



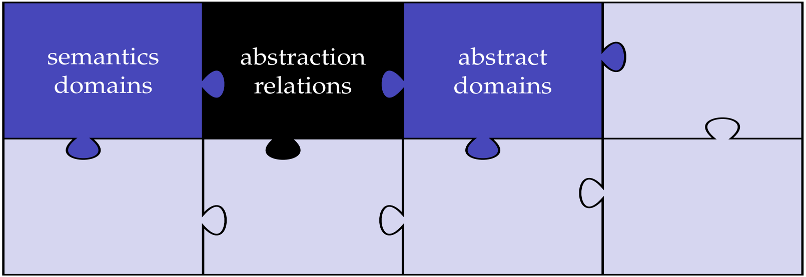

```
Inductive type : Set :=
      | ReferenceType (rt : refType)
      | PrimitiveType (pt: primitiveType)
  with refType :Set :=
      | ArrayType (typ:type)
      | ClassType  (ct:ClassName)
      | InterfaceType (it:InterfaceName)
  with  primitiveType : Set :=
      | BOOLEAN  | BYTE | SHORT | INT.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

Specific challenges, e.g. interfaces

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

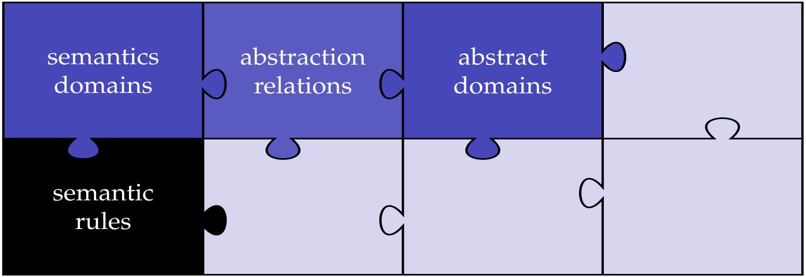
Both I and J are upper bounds for C and D, but they are incomparable.

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
```

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

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:

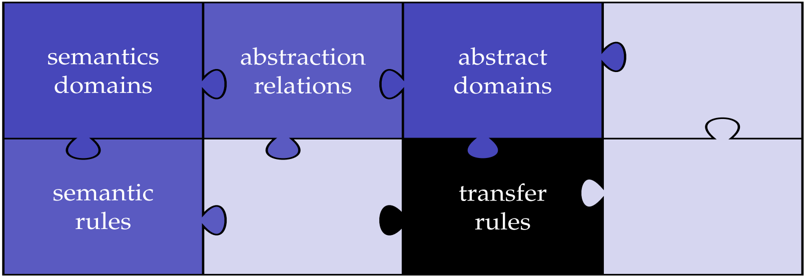
1. normal intra step
2. exception step
3. call step
4. return step

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

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

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

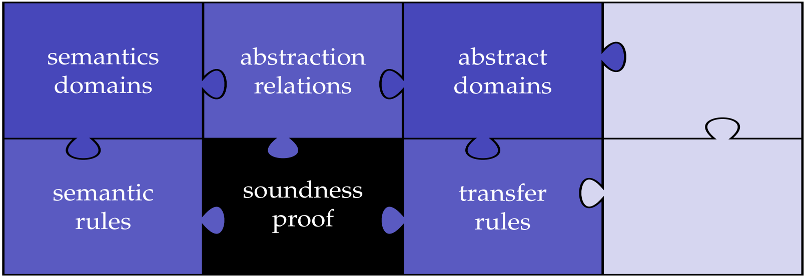
$$\frac{P[i] = \mathit{aload}\ n \quad rt(n) = \tau \quad \tau \prec \mathit{Object} \quad |st| + 1 \leqslant \mathit{Mstack}}{rt, st \Rightarrow rt, \tau :: st}$$

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

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

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

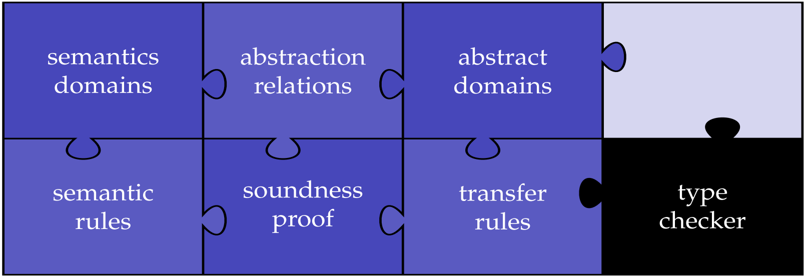
Verified bytecode verification



If $s \leadsto s'$ 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

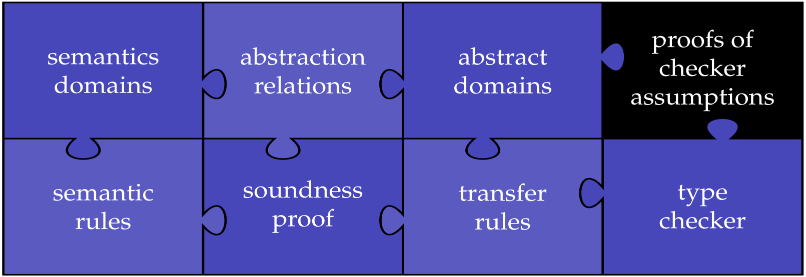
Verified bytecode verification



From declarative definition of typable program to type checker

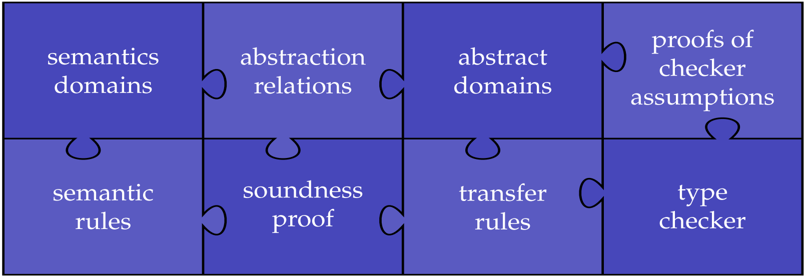
- rely on generic construction
- . . . but requires discharging hypotheses!

Verified bytecode verification



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

Verified bytecode verification



Final results

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

- 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

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

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

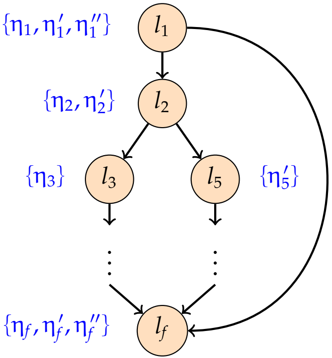
$$\begin{array}{rcl} i'' & = & min(i, i') \\ j'' & = & max(j, j') \end{array}$$

- Abstract semantics of addition

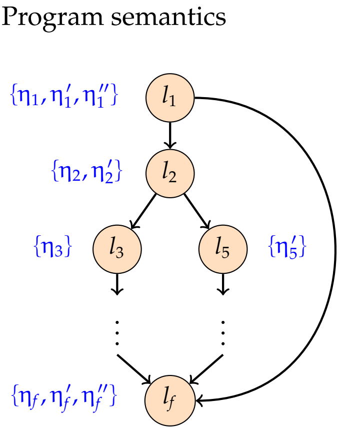
$$[i, j] + [i', j'] = [i + i', j + j']$$

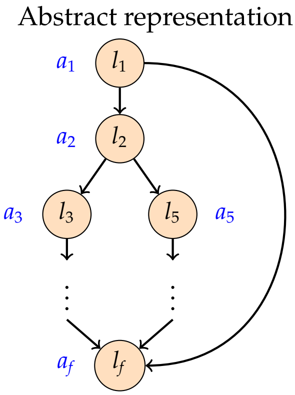
Concrete vs abstract semantics

Program semantics

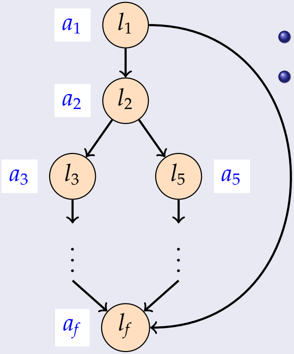


Concrete vs abstract semantics





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 $\hat{S} : \mathcal{P} \rightarrow \mathcal{A}$

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

  - 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 $\langle\{\gamma(a_1) \ldots \gamma(a_n)\}, c\rangle$ 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:

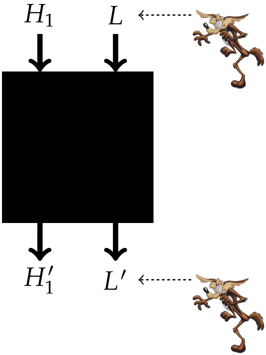
$$\frac{c : 0}{\mathsf{while}\ b\ \mathsf{do}\ c : 0}$$

One can enforce a similar constraint for bytecode using widening

- A program is bounded iff for every i s.t. $P[i] = \mathsf{new}, \mathsf{newarray}$, 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

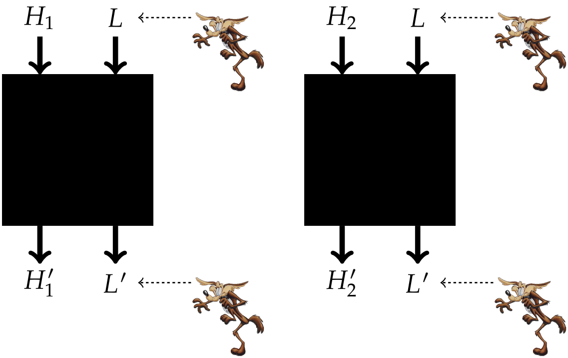
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidential   Low = public

Non-interference

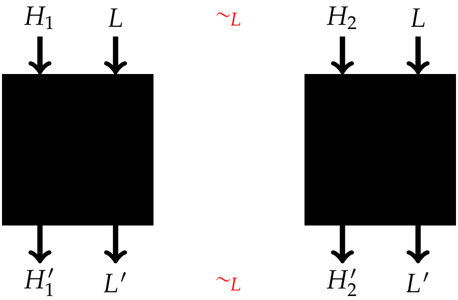
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



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:

$$
\begin{array}{lcll}
\mathit{instr} & ::= & \textsf{prim}\ op & \text{primitive operation} \\
& | & \textsf{push}\ v & \text{push value on top of stack} \\
& | & \textsf{load}\ x & \text{load value of } x \text{ on stack} \\
& | & \textsf{store}\ x & \text{store top of stack in } x \\
& | & \textsf{if}\ j & \text{conditional jump} \\
& | & \textsf{goto}\ j & \text{unconditional jump} \\
& | & \textsf{return} & \text{return}
\end{array}
$$

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

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

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

Semantics: rules

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

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

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

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

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

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

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

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

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 $i, j \in \mathcal{P}$
  - $i \mapsto j \Rightarrow \exists st'.\, i \vdash S_i \Rightarrow st' \wedge st' \leqslant S_j$;
  - $i \mapsto \Rightarrow \ i \vdash S_i \Rightarrow$

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

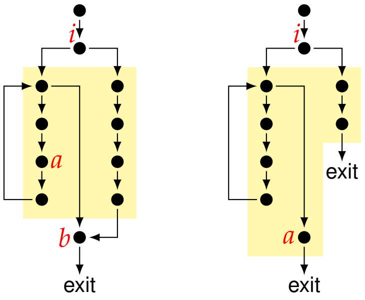
A program point j is in a *control dependence region* of a branching point i if

- 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 $region \in \mathcal{P} \to \wp(\mathcal{P})$ and $jun \in \mathcal{P} \rightharpoonup \mathcal{P}$.

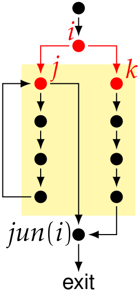
SOAP1: for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in region(i)$ or $k = jun(i)$;

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.

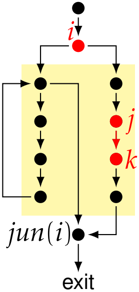


CDR soundness is ensured by local conditions (instead of path properties) using *region* $\in \mathcal{P} \to \wp(\mathcal{P})$ and *jun* $\in \mathcal{P} \rightharpoonup \mathcal{P}$.

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), $k \in \mathit{region}(i)$ or $k = \mathit{jun}(i)$;

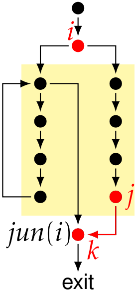
SOAP2: for all program points i, j, k, if $j \in \mathit{region}(i)$ and $j \mapsto k$, then either $k \in \mathit{region}(i)$ or $k = \mathit{jun}(i)$;

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



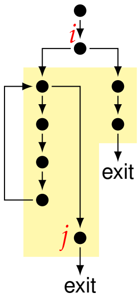
CDR soundness is ensured by local conditions (instead of path properties) using $region \in \mathcal{P} \to \wp(\mathcal{P})$ and $jun \in \mathcal{P} \rightharpoonup \mathcal{P}$.

- 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), $k \in \mathit{region}(i)$ or $k = \mathit{jun}(i)$;
- SOAP2: for all program points i, j, k, if $j \in \mathit{region}(i)$ and $j \mapsto k$, then either $k \in \mathit{region}(i)$ or $k = \mathit{jun}(i)$;
- SOAP3: for all program points i, j, if $j \in \mathit{region}(i)$ and $j \mapsto$ then $\mathit{jun}(i)$ is undefined.



Transfer rules

$$\frac{P[i] = \mathsf{push}\ n}{i \vdash st \Rightarrow se(i) :: st} \qquad \frac{P[i] = \mathsf{binop}\ op}{i \vdash k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2) :: st}$$

$$\frac{P[i] = \mathsf{load}\ x}{i \vdash st \Rightarrow (\Gamma(x) \sqcup se(i)) :: st} \qquad \frac{P[i] = \mathsf{store}\ x \qquad se(i) \sqcup k \leqslant \Gamma(x)}{i \vdash k :: st \Rightarrow st}$$

$$\frac{P[i] = \mathsf{goto}\ j}{i \vdash st \Rightarrow st} \qquad \frac{P[i] = \mathsf{return} \qquad se(i) \sqcup k \leqslant k_r}{i \vdash k :: st \Rightarrow}$$

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

State equivalence

Unwinding lemmas focus on state equivalence $\sim_L$.

State equivalence

$\langle\langle i, \rho, s\rangle\rangle \sim_L \langle\langle i', \rho', s'\rangle\rangle$ if:

- Memory equivalence $\rho \sim_L \rho'$
- Operand stack equivalence $s \overset{i,i'}{\sim}_L s'$ (defined w.r.t. S)

State equivalence

Unwinding lemmas focus on state equivalence $\sim_L$.

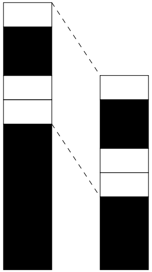
State equivalence

$\langle\langle i, \rho, s \rangle\rangle \sim_L \langle\langle i', \rho', s' \rangle\rangle$ if:

- Memory equivalence $\rho \sim_L \rho'$
- Operand stack equivalence $s \overset{i,i'}{\sim}_L s'$ (defined w.r.t. S)

Operand stack equivalence $s \overset{i,i'}{\sim}_L s'$ is defined w.r.t. S_i and $S_{i'}$:

- High stack positions in black
- Require that both stacks coincide, except in their lowest black portion



Soundness

If $S \vdash P$ (w.r.t. *se* and *cdr*) then P is non-interfering.

Direct application of

- Low (locally respects) unwinding lemma:
  If $s \sim_L s'$ and $s \leadsto t$ and $s' \leadsto t'$, then $t \sim_L t'$, provided $s \cdot pc = s' \cdot pc$
- High (step consistent) unwinding lemma:
  If $s \sim_L s'$ and $s \leadsto t$ and then $t \sim_L t'$, provided $s \cdot pc = i$ is a high program point and S_i is high and *se* is well-formed
- Gluing lemmas for combining high and low unwinding lemmas (extensive use of SOAP properties)
- Monotonicity lemmas

Compatibility with lightweight verification

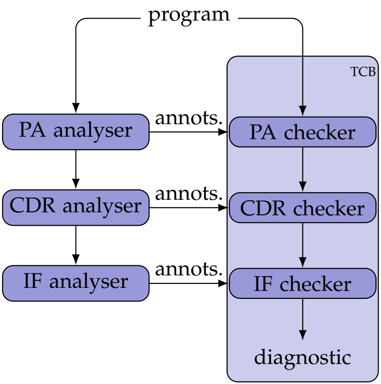
The type system:

- is compatible with lighweight bytecode verification
- code provided with
  - regions (verified by a region checker)
  - security environment
  - type information at junction points

Adding objects, exceptions and methods

Main issues:

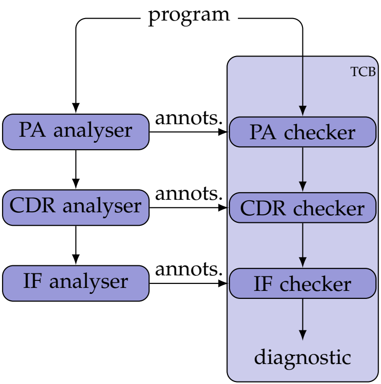
- objects
  (heap equivalence, allocator)
- exceptions
  (loss of precision)
- methods (extended signatures)



Adding objects, exceptions and methods

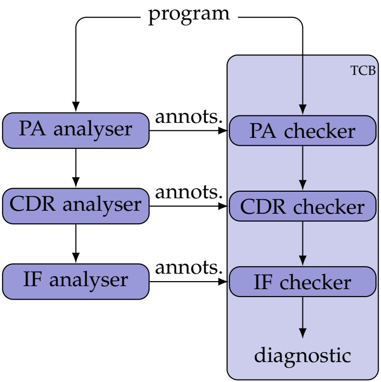
Three successive phases:

1. the PA (pre-analyse) analyser computes information to reduce the control flow graph.
2. the CDR analyser computes *control dependence regions* (to deal with implicit flows)
3. the IF (Information Flow) analyser computes for each program point a *security environment* and a *stack type*



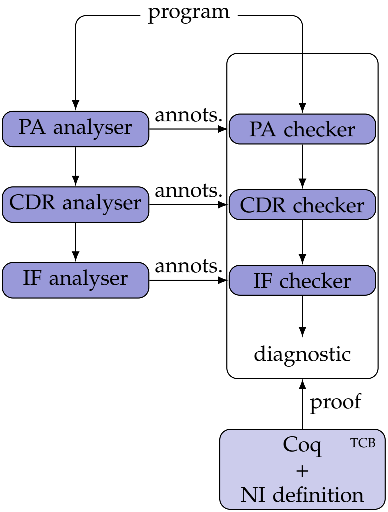
Adding objects, exceptions and methods

- Each phase corresponds to a pair analyser/checker
- Trusted Computed Base (TCB) is reduced to the checkers
- Moreover, since we prove these checkers in Coq, TCB is in fact relegated to Coq and the formal definition of non-interference.



Adding objects, exceptions and methods

- Each phase corresponds to a pair analyser/checker
- Trusted Computed Base (TCB) is reduced to the checkers
- Moreover, since we prove these checkers in Coq, TCB is in fact relegated to Coq and the formal definition of non-interference.



Pre-analyses

Branching is a major source of imprecision in an information flow static analysis.

The PA (pre-analyse) analyser computes information that is used to reduce the control flow graph and to detect branches that will never be taken.

- null pointers (to predict unthrowable null pointer exceptions),
- classes (to predict target of throws instructions),
- array accesses (to predict unthrowable out-of-bounds exceptions),
- exceptions (to over-approximate the set of throwable exceptions for each method)

Such analyses (and their respective certified checkers) can be developed using *certified abstract interpretation*.

Information flow type system

Type annotations required on programs:

- $ft : \mathcal{F} \to \mathcal{S}$ attaches security levels to fields,
- $at : \mathcal{M} \times \mathcal{P} \rightharpoonup \mathcal{S}$ attaches security levels to contents of arrays at their creation point
- each method posseses one (or several) signature(s):

$$\vec{k_v} \xrightarrow{k_h} \vec{k_r}$$

  - $\vec{k_v}$ provides the security level of the method parameters (and local variables),
  - k_h: effect of the method on the heap,
  - $\vec{k_r}$ is a record of security levels of the form $\{n : k_n, e_1 : k_{e_1}, \ldots e_n : k_{e_n}\}$
    - k_n is the security level of the return value (normal termination),
    - k_i is the security level of each exception e_i that might be propagated by the method

Example

```
int m(boolean x,C y) throws C {
  if (x) {throw new C();}
  else {y.f = 3;};
  return 1;
}
```

1	load x
2	if 5
3	new C
4	throw
5	load y
6	push 3
7	putfield f
8	push 1
9	return

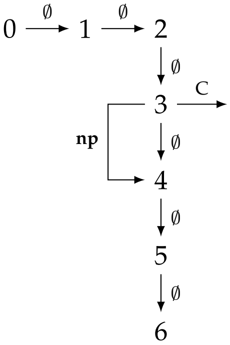
$$m : (x : L,\ y : H) \xrightarrow{H} \{\mathsf{n} : H,\ C : L,\ \mathbf{np} : H\}$$

- $k_h = H$: no side effect on low fields ,
- $\vec{k_r}[n] = H$: result depends on y,
- termination by an exception C doesn't depend on y,
- but termination by a null pointer exception does.

Fine grain exceptions handling : example

```
try {z = o.m(x,y);} catch (NPE z) {}; t = 1;
```

0 : load o_L
1 : load y_H
2 : load x_L
3 : invokevirtual m
4 : store z_H
5 : push 1
6 : store t_L
handler : $[0,3]$, NullPointer $\rightarrow 4$



With only one level for all exceptions

- $[4,5,6]$ is a high region (depends on y_H): $t_L = 1$ is rejected

With our signature

- $[4,5,6]$ is a low region: $t_L = 1$ is accepted
- a region is now associated to a branching point and a step kind (normal step or exception step)

Typing judgment

General form

$$\frac{P[i] = ins \quad constraints}{\Gamma, ft, region, se, sgn, i \vdash^{\tau} st \Rightarrow st'}$$

Selected rules

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \vec{k_a'} \xrightarrow{k_h'} \vec{k_r'} \\ k \sqcup k_h \sqcup se(i) \leqslant k_h' \qquad k \leqslant \vec{k_a'}[0] \qquad \forall i \in [0, \mathsf{length}(\mathrm{st}_1) - 1],\ \mathrm{st}_1[i] \leqslant \vec{k_a'}[i+1] \\ e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \cup \{\mathbf{np}\} \qquad \forall j \in region(i,e),\ k \sqcup \vec{k_r'}[e] \leqslant se(j) \qquad \mathsf{Handler}(i,e) = t \end{array}}{\Gamma, region, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{e} \mathrm{st}_1 :: k :: \mathrm{st}_2 \Rightarrow (k \sqcup \vec{k_r'}[e]) :: \varepsilon}$$

$$\frac{P[i] = \mathsf{xastore} \qquad k_1 \sqcup k_2 \sqcup k_3 \leqslant k_e \qquad \forall j \in region(i, \emptyset),\ k_e \leqslant se(j)}{\Gamma, region, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k_1 :: k_2 :: k_3[k_e] :: \mathrm{st} \Rightarrow \mathsf{lift}_{k_e}(\mathrm{st})}$$

Formalization in Coq

```
| invokevirtual : forall i (mid:MethodSignature) st1 k1 st2,
    length st1 = length (METHODSIGNATURE.parameters (snd mid)) ->
    compat_type_st_lvt (virtual_signature p (snd mid) k1) (st1++L.Simple k1::st2) (1+(length st1)) ->
    k1 <= (virtual_signature p (snd mid) k1).(heapEffect) ->
    (forall j, region i None j ->
      L.join (join_list (virtual_signature p (snd mid) k1).(resExceptionType) (throwableBy p (snd mid)))
      k1 <= se j) ->
    compat_op (METHODSIGNATURE.result (snd mid)) (virtual_signature p (snd mid) k1).(resType) ->
    sgn.(heapEffect) <= (virtual_signature p (snd mid) k1).(heapEffect) ->
    texec i (Invokevirtual mid) None
    (st1++L.Simple k1::st2)
    (Some (lift k1
          (lift (join_list (virtual_signature p (snd mid) k1).(resExceptionType) (throwableBy p (snd mid)))
      (cons_option (join_op k1 (virtual_signature p (snd mid) k1).(resType)) st2))))
```

See the Coq development for 63 others typing rules...

Remarks on machine-checked proof

We have used the Coq proof assistant to

- to formally define non-interference definition,
- to formally define an information type system,
- to mechanically proved that typability enforces non-interference,
- to program a type checker and prove it enforces typability,
- to extract an Ocaml implementation of this type checker.

Structure of proofs

1. Itermediate semantics simplifies the intermediate definition of indistinguishability (call stacks),
2. Second intermediate semantics : annotated semantics with result of pre-analyses
   - the pre-analyse checker enforces that both semantics correspond
3. Implementation and correctness proof of the CDR checker
4. The information flow type system (and its corresponding type checker) enforce non-interference wrt. the annotated semantics.

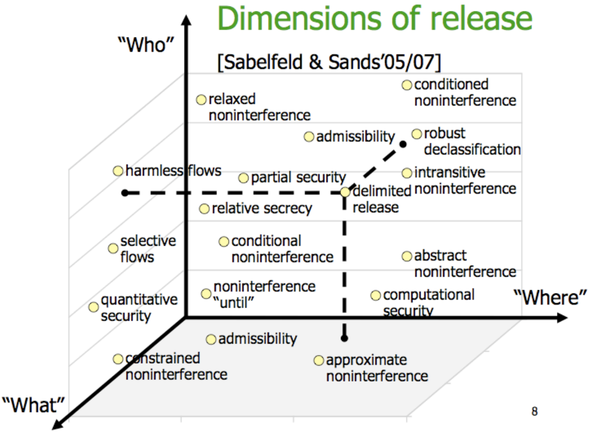
About 20,000 lines of definitions and proofs, inc. 3000 lines to define the JVM semantics

Towards realistic applications

Many features of missing to program realistic applications:

- declassification
- multi-threading
- flow sensitivity, polymorphism, etc

- Baseline policies (i.e. non-interference) are too restrictive in practice. Declassification policies allow intentional information release.
- Main dimensions: what, where, who



Information release for JVM

Goal is to define an information flow policy that:

- supports controlled release of information,
- that can be enforced efficiently,
- with a *modular proof of soundness*,
- instantiable to bytecode
- can reuse machine-checked proofs

Policy setting

- Setting is heavily influenced by non-disclosure, but allows declassification of a variable rather than of a principal.
- Policy is local to each program point:
  - modeled as an indexed family $(\sim_{\Gamma[i]})_{i \in \mathcal{P}}$ of relations on states
  - each $\sim_{\Gamma[i]}$ is symmetric and transitive
  - monotonicity of equivalence

$$\Gamma[i] \leqslant \Gamma[j] \wedge s \sim_{\Gamma[i]} t \Rightarrow s \sim_{\Gamma[j]} t$$

(properties hold when relations are induced by the security level of variables)

Delimited non-disclosure

P satisfies delimited non-disclosure (DND) iff $\mathsf{entry}\ \mathcal{R}\ \mathsf{entry}$, where $\mathcal{R}\ \subseteq \mathcal{P} \times \mathcal{P}$ satisfies for every $i, j \in \mathcal{P}$:

- if $i\ \mathcal{R}\ j$ then $j\ \mathcal{R}\ j$;
- if $i\ \mathcal{R}\ j$ then for all s_i, t_j and $s'_{i'}$ s.t.

$$s_i \leadsto s'_{i'} \wedge s_i \sim_{\Gamma[i]} t_j \wedge \mathrm{safe}(t_j)$$

  there exists $t'_{j'}$ such that:

$$t_j \leadsto^{\star} t'_{j'} \wedge s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t'_{j'} \wedge i'\ \mathcal{R}\ j'$$

Local policies vs. declassify statements

One could use a construction $\mathsf{declassify}\ (e)\ \mathsf{in}\ \{\ c\ \}$ and compute local policies from program syntax:

$$[l_1 := 0]^1\ ;\ \mathsf{declassify}\ (h)\ \mathsf{in}\ \{\ [l_2 := h]^2\ \}\ ;\ [l_3 := l_2]^3$$

yields

$$\begin{array}{l}
\Gamma[1](l_1) = \Gamma[1](l_2) = \Gamma[1](l_3) = L \\
\Gamma[1](h) = H \\
\Gamma[2](l_1) = \Gamma[2](l_2) = \Gamma[2](l_3) = L \\
\Gamma[2](h) = L \\
\Gamma[3] = \Gamma[1]
\end{array}$$

Where is what?

Declassification of expressions through fresh local variables:

$$\textsf{declassify}\ (h > 0)\ \textsf{in}\ \{\ [\textsf{if}\ (\ h > 0\ )\ \textsf{then}\ \{\ [l := 0]^2\ \}]^1\ \}$$

becomes

$$[h' := h > 0]^1\ ;$$
$$\textsf{declassify}\ (h')\ \textsf{in}\ \{\ [\textsf{if}\ (\ h'\ )\ \textsf{then}\ \{\ [l := 0]^3\ \}]^2\ \}$$

DND type system

- Given a NI type system $\Gamma, S, se \vdash i$; think as a shorthand for

$$\exists s_j.\ \Gamma[i], S, se \vdash S(i) \Rightarrow s_j \wedge s_j \leqslant S(j)$$

- Define a DND type system $(\Gamma[j])_{j \in \mathcal{P}}, S, se \vdash i$ as

$$\Gamma[i], S, se \vdash i$$

  (Note: not so easy for source languages)

- Program P is typable w.r.t. policy $(\Gamma[j])_{j \in \mathcal{P}}$ and type S iff for all i

$$\Gamma[i], S, se \vdash i$$

Soundness

If $(\Gamma[j])_{j \in \mathcal{P}}, S, se \vdash P$ then P satisfies DND.

- Policies must respect no creep up, ie $\Gamma[i](x) \leqslant \Gamma[\mathsf{entry}](x)$

Unwinding+Progress

- Unwinding: if $\Gamma, S \vdash_{NI} i$ then

$$(s_i \sim_\Gamma t_i \wedge s_i \rightsquigarrow s'_{i'} \wedge t_i \rightsquigarrow t'_{j'}) \Rightarrow s'_{i'} \sim_\Gamma t'_{j'}$$

- Progress: if i is not an exit point and $\mathrm{safe}(s_i)$ then there exists t s.t. $s_i \rightsquigarrow t$

$$\left.\begin{array}{c} (\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P \\ s_i \sim_{\Gamma[i]} t_i \\ s_i \rightsquigarrow s'_{i'} \\ \mathrm{safe}(t_i) \end{array}\right\} \Rightarrow \exists t'_{j'}.\, t_i \rightsquigarrow t'_{j'} \wedge s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t'_{j'}$$

High branches

- Unwinding: if $\Gamma, S \vdash_{NI} i$ and $H \leqslant se(i)$ then $(s_i \sim_\Gamma t_j \wedge s_i \rightsquigarrow s'_{i'}) \Rightarrow s'_{i'} \sim_\Gamma t_j$
- Exit from high loops: if i is a high branching point, then
  - $\mathsf{jun}(i)$ is defined
  - all executions entering $region(i)$ exit the region at $\mathsf{jun}(i)$
- No declassify in high context

$$H \leqslant se(i), se(j) \wedge i \mapsto j \Rightarrow \Gamma[i](x) = \Gamma[j](x)$$

$$\left.\begin{array}{c} (\Gamma[i])_{i \in \mathcal{P}}, S \vdash_{DND} P \\ i \text{ high branching} \\ j \in region(i) \\ \mathsf{safe}(s_j) \end{array}\right\} \exists s'_{\mathsf{jun}(i)}.\ s_j \rightsquigarrow^\star s'_{\mathsf{jun}(i)} \wedge s_j \sim_{\Gamma[\mathsf{entry}]} s'_{\mathsf{jun}(i)}$$

Bisimulation

$$\frac{}{i \,\mathcal{B}\, i} \qquad \frac{j \,\mathcal{B}\, i}{i \,\mathcal{B}\, j} \qquad \frac{i, j \in \mathit{region}(k) \cup \{\mathsf{jun}(k)\} \qquad \mathit{se}(k) = H}{i \,\mathcal{B}\, j}$$

- If $i, j \in \mathit{region}(k)$ for some k s.t. $H \leqslant \mathit{se}(k)$.
  Assume $s_i \sim_{\Gamma[i]} t_j$, and $s_i \rightsquigarrow s'_{i'}$.
  Choose $t' = t$.
  By unwinding and monotonicity, $s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t_j$.
  By exit through junction, either $i' \in \mathit{region}(k)$ or $i' = \mathsf{jun}(k)$.
- If $j \in \mathit{region}(k)$ and $i = \mathsf{jun}(k)$ for some k s.t. $H \leqslant \mathit{se}(k)$.
  ...

Laundering attacks

$$[h := h']^1 \ ; \ \textsf{declassify}\ (h)\ \textsf{in}\ \{\ [l := h]^2\ \}$$

- Such programs are insecure w.r.t. policies such as localized delimited release.
- It is possible to define a simple effect system that prevents laundering attacks:
  - judgments are of the form $\vdash_{LA} c : U, V$
  - U is the set of assigned variables
  - V is the set of declassified variables

Concurrency

- Mobile code applications often exploit concurrency
- Concurrent execution of secure sequential programs is not necessarily secure:

$$\mathsf{if}(h > 0)\{\mathsf{skip};\mathsf{skip}\}\{\mathsf{skip}\}; l := 1 \qquad \| \qquad \mathsf{skip};\mathsf{skip}; l := 2$$

- Security of multi-threaded programs can be achieved:
  - by imposing strong security conditions on programs
  - by relying on secure schedulers

Secure schedulers

A secure scheduler selects the thread to be executed in function of the security environment:

- the thread pool is partitioned into low, high, and hidden threads
- if a thread is currently executing a high branch, then only high threads are scheduled
- if the program counter of the last executed thread becomes high (resp. low), then the thread becomes hidden or high (resp. low)
- the choice of a low thread only depends on low history

Round-robin schedulers are secure, provided they take over control when threads become high/low/hidden

Multi-threaded language

- New instruction $\mathsf{start}\ i$
- States $\langle\!\langle \rho, \lambda \rangle\!\rangle$ where λ associates to each active thread a pair $\langle\!\langle i, s \rangle\!\rangle$.
- Semantics $s, h \leadsto s'$:
  - h is an history
  - implicitly parameterized by scheduler (modeled as function pickt from states and histories to threads) and security environment
  - most rules inherited from sequential fragment

$$\frac{\begin{array}{c}\mathsf{pickt}(\langle\!\langle \rho, \lambda \rangle\!\rangle, h) = ctid \\ \lambda(ctid) = \langle\!\langle i, s \rangle\!\rangle \\ P[i] \neq \mathsf{start}\ k \\ \langle\!\langle i, \rho, s \rangle\!\rangle \leadsto_{\mathrm{seq}} \langle\!\langle i', \rho', s' \rangle\!\rangle\end{array}}{\langle\!\langle \rho, \lambda \rangle\!\rangle, h \leadsto \langle\!\langle \rho', \lambda' \rangle\!\rangle}$$

where

$$\lambda'(tid) = \begin{cases} \langle\!\langle i', s' \rangle\!\rangle & \text{if } tid = ctid \\ \lambda(tid) & \text{otherwise} \end{cases}$$

$$\frac{\begin{array}{c}\mathsf{pickt}(\langle\!\langle \rho, \lambda \rangle\!\rangle, h) = ctid \\ \lambda(ctid) = \langle\!\langle i, s \rangle\!\rangle \\ P[i] = \mathsf{start}\ pc \\ ntid \text{ fresh}\end{array}}{\langle\!\langle \rho, \lambda \rangle\!\rangle, h \leadsto \langle\!\langle \rho', \lambda' \rangle\!\rangle}$$

where

$$\lambda'(tid) = \begin{cases} \langle\!\langle pc, \epsilon \rangle\!\rangle & \text{if } tid = ntid \\ \lambda(tid) & \text{otherwise} \end{cases}$$

Policy and type system

- Policy is similar to sequential fragment
- Transfer rules inherited from sequential fragment

$$\frac{P[i] \neq \mathsf{start}\ j \qquad i \vdash_{\mathrm{seq}} st \Rightarrow st'}{i \vdash st \Rightarrow st'} \qquad \frac{P[i] = \mathsf{start}\ j \qquad se(i) \leqslant se(j)}{i \vdash st \Rightarrow st}$$

- Type system similar to sequential fragment. As in bytecode verification, each thread is verified in isolation.
  - If $P[i] = \mathsf{start}\ j$ we do not have $i \mapsto j$
- Assume the scheduler is secure, type soundness can be lifted from sequential language

Type-preserving compilation

- Source type systems offer tools for developing safe/secure applications, but does not directly address mobile code
- Bytecode verifiers provides safety/security assurance to users
- Relating both type systems ensure:
  - applications can be deployed in a mobile code architecture that delivers the promises of the source type system
  - enhanced safety/security architecture can benefit from tools for developing applications that meet the policy it enforces

Compiler correctness

The compiler is semantics-preserving (terminating runs, input/output behavior)

$$P, \mu \Downarrow \nu, v \quad \Rightarrow \quad [\![P]\!], \mu \Downarrow \nu, v$$

Thus source programs satisfy an input/output property iff their compilation does

$$\begin{array}{l} \forall P, \phi, \psi, \mu, \nu, v. \\ (\phi(\mu) \Rightarrow P, \mu \Downarrow \nu, v \Rightarrow \psi(\mu, \nu, v)) \\ \Rightarrow (\phi(\mu) \Rightarrow [\![P]\!], \mu \Downarrow \nu, v \Rightarrow \psi(\mu, \nu, v)) \end{array}$$

But are typable programs compiled into typable programs?

$$\forall P, \vdash P \Longrightarrow \exists S.\ S, \vdash [\![P]\!]$$

Yes for JVM typing, no in general

Loss of information

Using the sign abstraction

$$x := 1; y := x - x$$

yields

$$y = zero$$

But

push 1
store x
load x
load x
op $-$
store y

yields

$$y = \top$$

Solutions:

- Change lattice
- Decompile expressions

Source language: While

A program is a command:

$$
\begin{array}{llcll}
\text{commands} & c & ::= & x := e & \text{assignment} \\
& & \mid & \mathsf{if}(e)\{c\}\{c\} & \text{conditional} \\
& & \mid & \mathsf{while}(e)\{c\} & \text{loop} \\
& & \mid & c; c & \text{sequence} \\
& & \mid & \mathsf{skip} & \text{skip} \\
& & \mid & \mathsf{return}\ e & \text{return value}
\end{array}
$$

Semantics is standard:

- States are pairs $\langle\!\langle c, \rho \rangle\!\rangle$
- Small-step semantics $\langle\!\langle c, \rho \rangle\!\rangle \leadsto \langle\!\langle c', \rho' \rangle\!\rangle$ or $\langle\!\langle c, \rho \rangle\!\rangle \leadsto \langle\!\langle \nu, v \rangle\!\rangle$
- Evaluation semantics $c, \mu \Downarrow \langle\!\langle \nu, v \rangle\!\rangle$ iff $c, \mu \leadsto^{\star} \langle\!\langle \nu, v \rangle\!\rangle$

Information flow type system

- Security policy $\Gamma : \mathcal{X} \to \mathcal{S}$ and k_{ret}
- Volpano-Smith security type system

$$\frac{e : k \quad k \sqcup pc \leqslant \Gamma(x)}{[pc] \vdash x := e} \qquad \frac{[k] \vdash c \qquad [k] \vdash c'}{[pc] \vdash c; c'}$$

$$\frac{e : k \quad [k] \vdash c_1 \quad [k] \vdash c_2}{[pc] \vdash \mathsf{if}(e)\{c_1\}\{c_2\}} \qquad \frac{e : k \quad [k] \vdash c}{[pc] \vdash \mathsf{while}(e)\{c\}}$$

$$\frac{e : k \quad k \sqcup pc \leqslant k_{\mathrm{ret}}}{[pc] \vdash \mathsf{return}\ e} \qquad \frac{}{[pc] \vdash \mathsf{skip}}$$

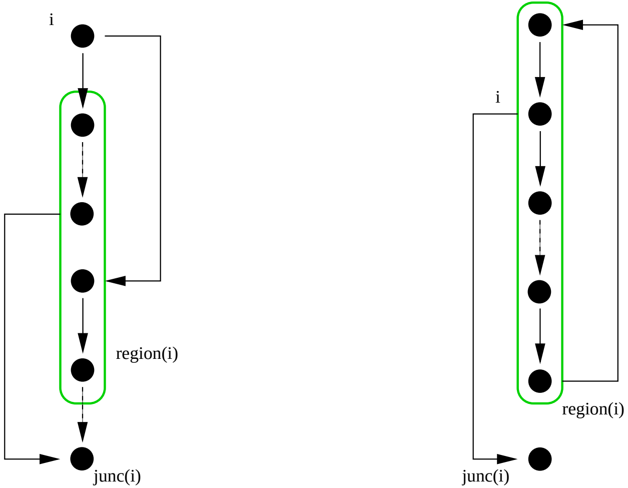
plus subtyping rules

$$\frac{[pc] \vdash c \quad pc' \leqslant pc}{[pc'] \vdash c'} \qquad \frac{e : k \quad k \leqslant k'}{e : k'}$$

Compiling statements

$$
\begin{array}{rcl}
[\![x]\!] & = & \mathsf{load}\ x \\
[\![v]\!] & = & \mathsf{push}\ v \\
[\![e_1\ op\ e_2]\!] & = & [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{binop}\ op \\
\\
k : [\![x := e]\!] & = & [\![e]\!];\ \mathsf{store}\ x \\
k : [\![i_1; i_2]\!] & = & k : [\![i_1]\!]; k_2 : [\![i_2]\!] \\
\text{where } k_2 & = & k + |[\![i_1]\!]| \\
\\
k : [\![\mathsf{return}\ e]\!] & = & [\![e]\!];\ \mathsf{return} \\
\\
k : [\![\mathsf{if}(e_1\ cmp\ e_2)\{i_1\}\{i_2\}]\!] & = & [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i_1]\!];\ \mathsf{goto}\ l;\ k_2 : [\![i_2]\!] \\
\text{where } k_1 & = & k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\
k_2 & = & k_1 + |[\![i_1]\!]| + 1 \\
l & = & k_2 + |[\![i_2]\!]| \\
\\
k : [\![\mathsf{while}(e_1\ cmp\ e_2)\{i\}]\!] & = & [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i]\!];\ \mathsf{goto}\ k \\
\text{where } k_1 & = & k + |[\![e_2]\!]| + |[\![e_1]\!]| + 1 \\
k_2 & = & k_1 + |[\![i]\!]| + 1
\end{array}
$$

Compiling control dependence regions



Compiling security environment

if$(y_H)\{x := 1\}\{x := 2\};$
$x' := 3;$
return 2

load y_H	L	
if 6	L	
push 1	H	$\in region(2)$
store x	H	$\in region(2)$
goto 8	H	$\in region(2)$
push 2	H	$\in region(2)$
store x	H	$\in region(2)$
push 3	L	$jun(2)$
store x'	L	
push 2	L	
return	L	

Preservation of information flow types

If P is typable, then the extended compiler generates security environment, regions, and stack types at junction points, such that:

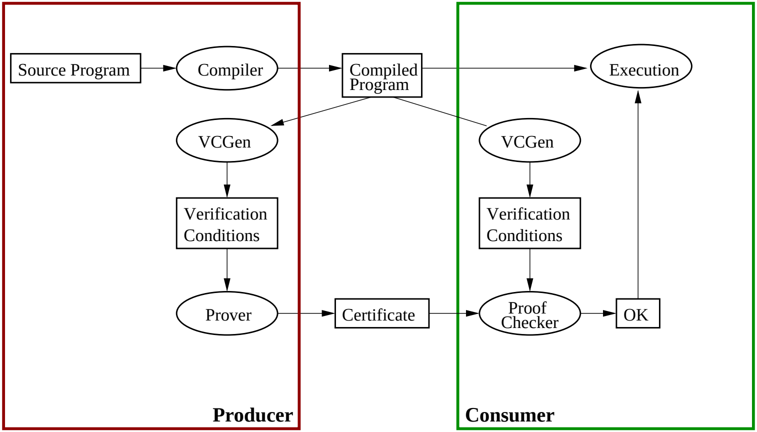
- regions satisfy SOAP and can be checked by region checker
- $[\![P]\!]$ can be verified by lightweight checker

The result also applies to

- concurrency (using naive rule for parallel composition)
- declassification

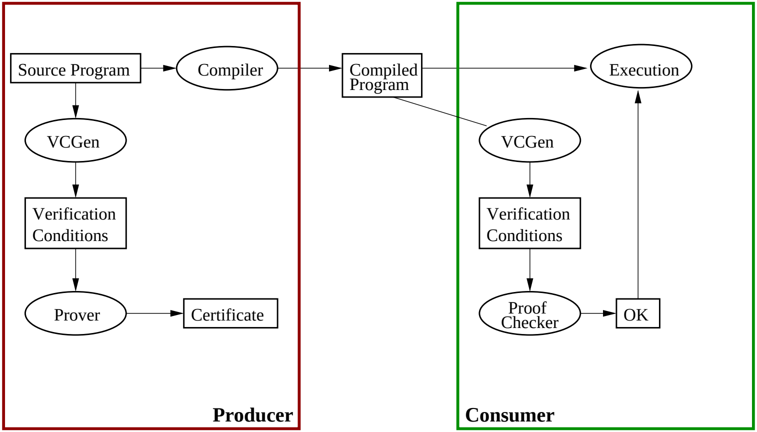
Motivation: source code verification

Traditional PCC



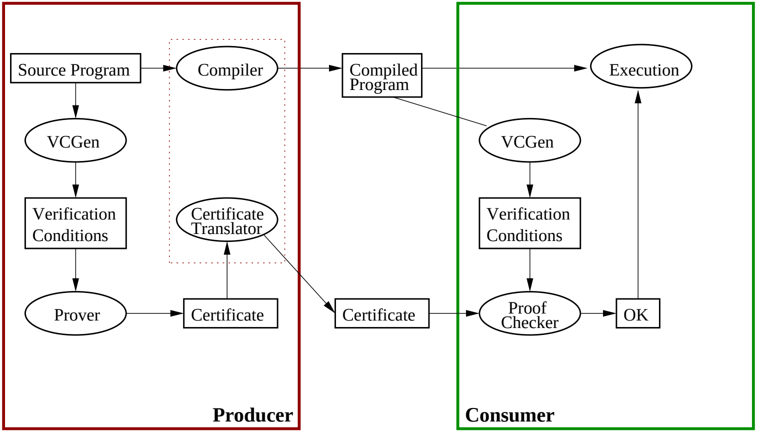
Motivation: source code verification

Source Code Verification

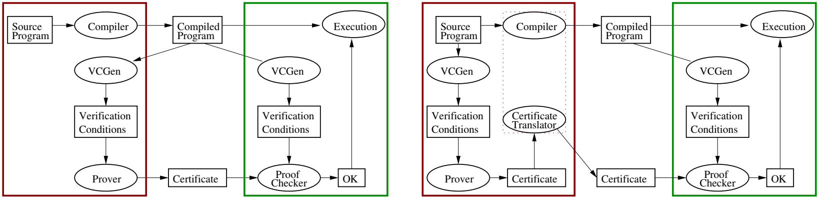


Motivation: source code verification

Certificate Translation



Certificate translation vs certifying compilation



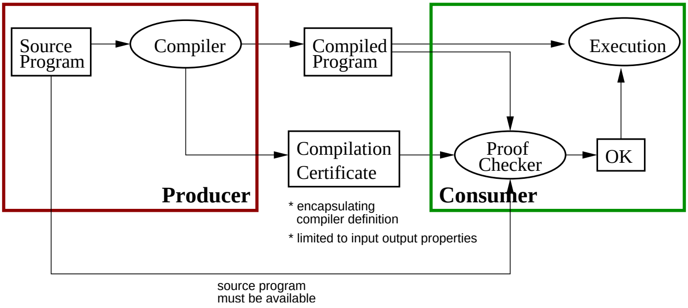
Conventional PCC		**Certificate Translation**
Automatically inferred invariants	**Specification**	Interactive
Automatic certifying compiler	**Verification**	Interactive source verification
Safety	**Properties**	Complex functional properties

Certificate translation vs certified compilation

Certified compilation aims at producing a proof term H such that

$$H : \forall P\ \mu\ \nu,\ P, \mu \Downarrow \nu \implies [\![P]\!], \mu \Downarrow \nu$$

Thus, we can build a proof term $H' : \{\phi\}[\![P]\!]\{\psi\}$ from H and $H_0 : \{\phi\}P\{\psi\}$



Program Specification

$\{pre\}$
ins_1
$\{\varphi_1\}$
ins_2
$\vdots$
$\{\varphi_2\}$
ins_k
$\{post\}$

- Assertions: formulae attached to a program point, characterizing the set of execution states at that point.
- Instructions are *possibly annotated*:

Possibly annotated instructions

$\overline{\mathsf{ins}} ::= \mathsf{ins} \mid \langle \varphi, \mathsf{ins} \rangle$

- A partially annotated program is a triple $\langle P, \Phi, \Psi \rangle$ s.t.
  - Φ is a precondition and Ψ is a postcondition
  - P is a sequence of possibly annotated instructions

Program Specification

$\{pre\}$
ins_1
$\{\varphi_1\}$
ins_2
$\vdots$
$\{\varphi_2\}$
ins_k
$\{post\}$

- Assertions: formulae attached to a program point, characterizing the set of execution states at that point.
- Instructions are *possibly annotated*:

Possibly annotated instructions

$\overline{\mathsf{ins}} ::= \mathsf{ins} \mid \langle \varphi, \mathsf{ins} \rangle$

- A partially annotated program is a triple $\langle P, \Phi, \Psi \rangle$ s.t.
  - Φ is a precondition and Ψ is a postcondition
  - P is a sequence of possibly annotated instructions

Program Specification

$\{pre\}$
ins_1
$\{\varphi_1\}$
ins_2
$\vdots$
$\{\varphi_2\}$
ins_k
$\{post\}$

- Assertions: formulae attached to a program point, characterizing the set of execution states at that point.
- Instructions are *possibly annotated*:

Possibly annotated instructions

$\overline{\mathsf{ins}} ::= \mathsf{ins} \mid \langle \varphi, \mathsf{ins} \rangle$

- A partially annotated program is a triple $\langle P, \Phi, \Psi \rangle$ s.t.
  - Φ is a precondition and Ψ is a postcondition
  - P is a sequence of possibly annotated instructions

Program Specification

$\{pre\}$
ins_1
$\{\varphi_1\}$
ins_2
$\vdots$
$\{\varphi_2\}$
ins_k
$\{post\}$

- Assertions: formulae attached to a program point, characterizing the set of execution states at that point.
- Instructions are *possibly annotated*:

Possibly annotated instructions

$\overline{\mathsf{ins}} ::= \mathsf{ins} \mid \langle \varphi, \mathsf{ins} \rangle$

- A partially annotated program is a triple $\langle P, \Phi, \Psi \rangle$ s.t.
  - Φ is a precondition and Ψ is a postcondition
  - P is a sequence of possibly annotated instructions

Program Specification

$\{pre\}$
ins_1
$\{\varphi_1\}$
ins_2
$\vdots$
$\{\varphi_2\}$
ins_k
$\{post\}$

- Assertions: formulae attached to a program point, characterizing the set of execution states at that point.
- Instructions are *possibly annotated*:

Possibly annotated instructions

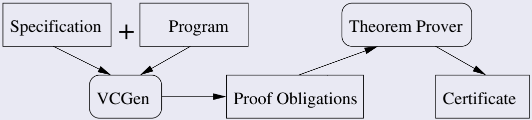
$$\overline{\mathsf{ins}} ::= \mathsf{ins} \mid \langle \varphi, \mathsf{ins} \rangle$$

- A partially annotated program is a triple $\langle P, \Phi, \Psi \rangle$ s.t.
  - Φ is a precondition and Ψ is a postcondition
  - P is a sequence of possibly annotated instructions

Building a certificate

Certification of annotated programs is performed in three steps

1. A verification condition generator fully annotates the program, and extracts a set of verification conditions (a.k.a. proof obligations)
2. verification conditions are discharged interactively
3. a certificate is built from proofs of verification conditions



Weakest precondition calculus

Computes an assertion for a given program node **only if** the corresponding assertion has been already computed for all successor nodes

Weakest precondition calculus

Computes an assertion for a given program node **only if** the corresponding assertion has been already computed for all successor nodes

Sufficiently annotated program

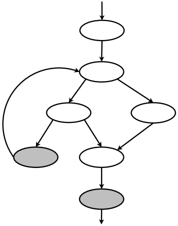
All infinite paths must go through an annotated program point

Weakest precondition calculus

Computes an assertion for a given program node **only if** the corresponding assertion has been already computed for all successor nodes

Sufficiently annotated program

All infinite paths must go through an annotated program point

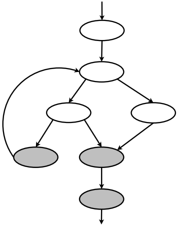


Weakest precondition calculus

Computes an assertion for a given program node **only if** the corresponding assertion has been already computed for all successor nodes

Sufficiently annotated program

All infinite paths must go through an annotated program point



Weakest precondition $\mathsf{wp}_{\mathcal{L}}(k)$ of program point k

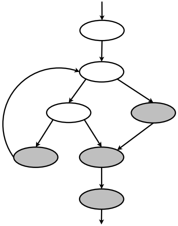
$$
\begin{array}{lcll}
\mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise}
\end{array}
$$

Weakest precondition calculus

Computes an assertion for a given program node **only if** the corresponding assertion has been already computed for all successor nodes

Sufficiently annotated program

All infinite paths must go through an annotated program point



Weakest precondition $\mathsf{wp}_{\mathcal{L}}(k)$ of program point k

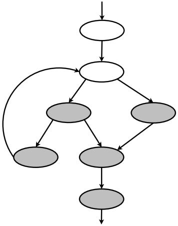
$$
\begin{array}{lcll}
\mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise}
\end{array}
$$

Weakest precondition calculus

Computes an assertion for a given program node **only if** the corresponding assertion has been already computed for all successor nodes

Sufficiently annotated program

All infinite paths must go through an annotated program point



Weakest precondition $\mathsf{wp}_{\mathcal{L}}(k)$ of program point k

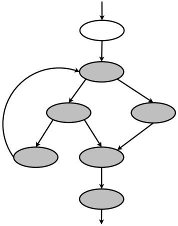
$$\begin{array}{lcll} \mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\ \mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise} \end{array}$$

Weakest precondition calculus

Computes an assertion for a given program node **only if** the corresponding assertion has been already computed for all successor nodes

Sufficiently annotated program

All infinite paths must go through an annotated program point



Weakest precondition $\mathsf{wp}_{\mathcal{L}}(k)$ of program point k

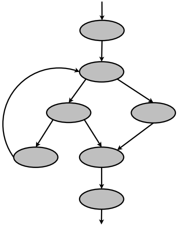
$$
\begin{array}{lcll}
\mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise}
\end{array}
$$

Weakest precondition calculus

Computes an assertion for a given program node **only if** the corresponding assertion has been already computed for all successor nodes

Sufficiently annotated program

All infinite paths must go through an annotated program point



Weakest precondition $\mathsf{wp}_{\mathcal{L}}(k)$ of program point k

$$\begin{array}{lcll} \mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\ \mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise} \end{array}$$

Assertions

- Annotations do not refer to stacks
- Intermediate assertions may do so

Assertions

- Annotations do not refer to stacks
- Intermediate assertions may do so

Assertions

- Annotations do not refer to stacks
- Intermediate assertions may do so

$\{true\}$
push 5
store x
$\{x = 5\}$

Assertions

- Annotations do not refer to stacks
- Intermediate assertions may do so

$\{true\}$
push 5
store x $\quad os[\top] = 5$
$\{x = 5\}$

Assertions

- Annotations do not refer to stacks
- Intermediate assertions may do so

$\{true\}$	
push 5	$5 = 5$
store x	$os[\top] = 5$
$\{x = 5\}$	

Assertions

- Annotations do not refer to stacks
- Intermediate assertions may do so

$\{true\}$	
push 5	$5 = 5$
store x	$os[\top] = 5$
$\{x = 5\}$	

Stack indices

$$k \quad ::= \quad \top \mid \top - i$$

Expressions

$$e \quad ::= \quad \mathsf{res} \mid x^\star \mid x \mid c \mid e\ op\ e \mid os[k]$$

Assertions

$$\begin{aligned} \phi \quad ::= \quad & e\ cmp\ e \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \\ & \forall x.\ \phi \mid \exists x.\ \phi \end{aligned}$$

Weakest precondition

- if $P[k] = \textsf{push}\ n$ then
$$\textsf{wp}_i(k) = \textsf{wp}_{\mathcal{L}}(k+1)[n/os[\top], \top/\top - 1]$$
- if $P[k] = \textsf{binop}\ op$ then
$$\textsf{wp}_i(k) = \textsf{wp}_{\mathcal{L}}(k+1)[os(\top - 1)\ op\ os[\top]/os[\top], \top - 1/\top]$$
- if $P[k] = \textsf{load}\ x$ then
$$\textsf{wp}_i(k) = \textsf{wp}_{\mathcal{L}}(k+1)[x/os[\top], \top/\top - 1]$$
- if $P[k] = \textsf{store}\ x$ then
$$\textsf{wp}_i(k) = \textsf{wp}_{\mathcal{L}}(k+1)[os[\top]/x, \top - 1/\top]$$
- if $P[k] = \textsf{if}\ cmp\ l$ then
$$\begin{aligned}\textsf{wp}_i(k) = {} & (os[\top - 1]\ cmp\ os[\top] \Rightarrow \textsf{wp}_{\mathcal{L}}(k+1)[\top - 2/\top]) \\ & \wedge(\neg(os[\top - 1]\ cmp\ os[\top]) \Rightarrow \textsf{wp}_{\mathcal{L}}(l)[\top - 2/\top])\end{aligned}$$
- if $P[k] = \textsf{goto}\ l$ then $\textsf{wp}_i(k) = \textsf{wp}_{\mathcal{L}}(l)$
- if $P[k] = \textsf{return}$ then $\textsf{wp}_i(k) = \Psi[os[\top]/\textsf{res}]$

Verification conditions

Proof obligations $\mathsf{PO}(P, \Phi, \Psi)$

- Precondition implies the weakest precondition of entry point:

$$\Phi \Rightarrow \mathsf{wp}_{\mathcal{L}}(1)$$

- For all annotated program points ($P[k] = \langle \varphi, i \rangle$), the annotation φ implies the weakest precondition of the instruction at k:

$$\varphi \Rightarrow \mathsf{wp}_i(k)$$

An annotated program is correct if its verification conditions are valid.

Soundness

Define validity of assertions:

- $s \models \phi$
- $\mu, s \models \phi$ (shorthand $\mu, \nu \models \phi$ if ϕ does not contain stack indices)

If (P, Φ, Ψ) is correct, and

- $P, \mu \Downarrow \nu, v$
- $\mu \models \Phi$

then

$$\mu, \nu \models \Psi[{}^{v}\!/\!_{\mathsf{res}}]$$

Furthermore, all intermediate assertions are verified

Proof idea: if $s \rightsquigarrow s'$ and $s \cdot pc = k$ and $s' \cdot pc = k'$,

$$\mu, s \models \mathsf{wp}_i(k) \quad \Longrightarrow \quad \mu, s' \models \mathsf{wp}_{\mathcal{L}}(k')$$

Source language

- Same assertions, without stack expressions
- Annotated programs $(\mathcal{P}, \Phi, \Psi)$, with all loops annotated $\mathsf{while}_I(t)\{s\}$
- Weakest precondition

$$\frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{skip}, \mathrm{post}) = \mathrm{post}, \emptyset} \qquad \frac{}{\mathsf{wp}_{\mathcal{S}}(x := e, \mathrm{post}) = \mathrm{post}[e/x], \emptyset}$$

$$\frac{\mathsf{wp}_{\mathcal{S}}(i_t, \mathrm{post}) = \phi_t, \theta_t \quad \mathsf{wp}_{\mathcal{S}}(i_f, \mathrm{post}) = \phi_f, \theta_f}{\mathsf{wp}_{\mathcal{S}}(\mathsf{if}(t)\{i_t\}\{i_f\}, \mathrm{post}) = (t \Rightarrow \phi_t) \wedge (\neg t \Rightarrow \phi_t), \theta_t \cup \theta_f}$$

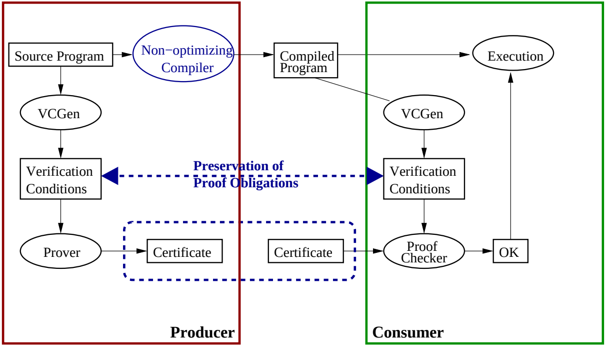
$$\frac{\mathsf{wp}_{\mathcal{S}}(i, I) = \phi, \theta}{\mathsf{wp}_{\mathcal{S}}(\mathsf{while}_I(t)\{i\}, \mathrm{post}) = I, \{I \Rightarrow ((t \Rightarrow \phi) \wedge (\neg t \Rightarrow \mathrm{post}))\} \cup \theta}$$

$$\frac{\mathsf{wp}_{\mathcal{S}}(i_2, \mathrm{post}) = \phi_2, \theta_2 \quad \mathsf{wp}_{\mathcal{S}}(i_1, \phi_2) = \phi_1, \theta_1}{\mathsf{wp}_{\mathcal{S}}(i_1; i_2, \mathrm{post}) = \phi_1, \theta_1 \cup \theta_2}$$

Preservation of proof obligations
Non-optimizing compiler

Syntactically equal proof obligations

$$\mathsf{PO}(P, \phi, \psi) = \mathsf{PO}([\![P]\!], \phi, \psi)$$



PPO: from (sequential) Java to JVM

We prove PPO for idealized, sequential fragments of Java and the JVM

Java vs JVM

- Statement language (obviously)
- Naming convention
- Basic types
- Compiler does simple optimizations

- Verification methods for Java programs must address known issues with objects, methods, exceptions.
- We use standard techniques: pre- and (exceptional) post-conditions, behavioral subtyping

Implementing a proof transforming compiler
(work by J. Charles and H. Lehner, using Mobius verification infrastructure)

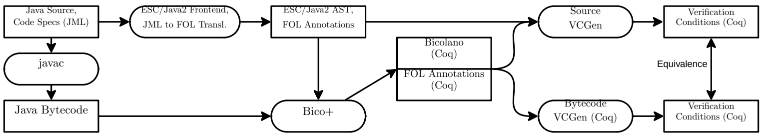
Reflective Proof Carrying Code

Programmed and formally verified a the verification condition generator against reference specification of sequential JVM

We have built a proof transforming compiler that

- generates for each annotated program a prelude and a set of VCs
- prove equivalence between source VCs and bytecode VCs

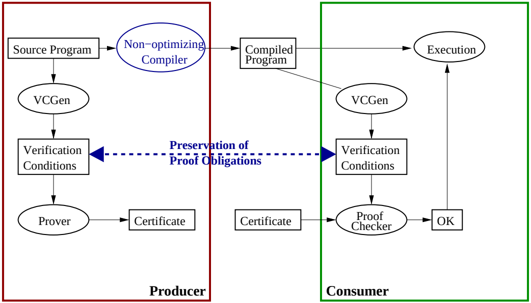
```
Lemma vc_equiv: vc_source <-> vc_bytecode.
```



The main tactic

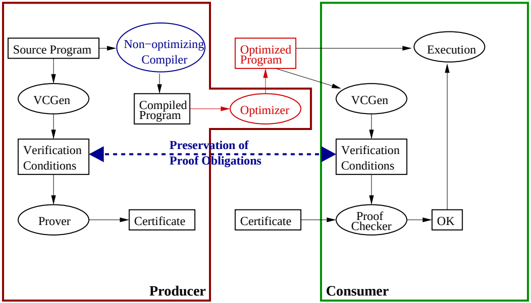
```
Ltac magickal :=
   repeat match goal with
   | [ |- forall lv: LocalVar.t, _ ] =>let lv := fresh "lv" in
                                            intro lv; mklvget lv 0%N
   | [ H: forall lv: LocalVar.t, _ |- _ ] => mklvupd MDom.LocalVar.empty 0%N
   | [ |- forall os: OperandStack.t, _ ] => intro
   | [ H: forall os: OperandStack.t, _ |- _ ] =>
              let H' := fresh "H" in (assert (H' := H OperandStack.empty); clear H)
   | [ H : forall y: Heap.t, _ |- forall x: Heap.t, _] =>
              let x := fresh "h" in
              (intro x; let H1 := fresh "H" in (assert (H1 := H x);
               clear H; try (clear x)))
   | [ H : forall y: Int.t, _ |- forall x: Int.t, _] =>
              let x := fresh "i" in (intro x; let H1 := fresh "H" in
                    (assert (H1 := H x); clear H; try (clear x)))
   | [ H : _ -> _ |- _ -> _] =>
              let A := fresh "H" in  (intros A; let H1 := fresh "H" in
                    (assert (H1 := H A); clear H; clear A))
   | [ H : _ /\ _ |- _ /\ _] =>let A := fresh "H" in
                                     let B := fresh "H" in
                                     (destruct H as (A, B); split; [clear B | clear A])
    end.
```

Optimizing Compilers



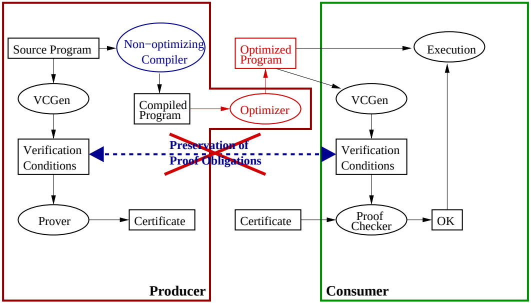
Proofs obligations might not be preserved

Optimizing Compilers



Proofs obligations might not be preserved

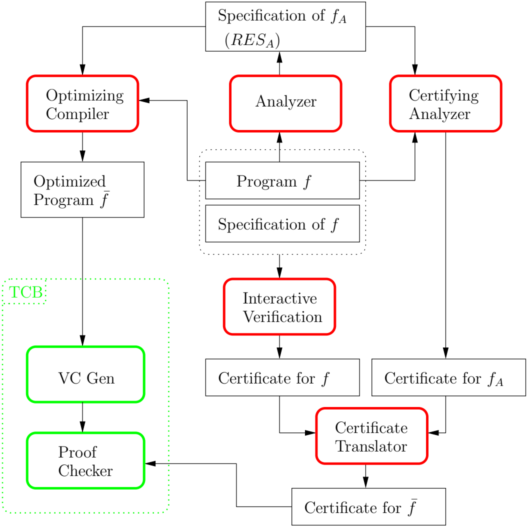
Optimizing Compilers



Proofs obligations might not be preserved

- annotations might need to be modified (e.g. constant propagation)
- certificates for analyzers might be needed (certifying analyzer)
- analyses might need to be modified (e.g. dead variable elimination)

Certificate Translation with Certifying Analyzers



Motivating example

$\{j = 0\}$

$i := 0;$

$x := b + i;$
$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$while(i! = n)$

$\quad i := c + i$

$\quad j := x * i;$

$endwhile;$
$\{n * b \leqslant j\}$

Program
+
Specification

Motivating example

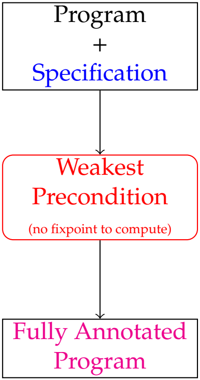
$\{j = 0\}$

$i := 0;$
$\{j = (b+i) * i \wedge b \leqslant (b+i) \wedge 0 \leqslant i\}$
$x := b + i;$
$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$while(i! = n)$

$\quad i := c + i$

$\quad j := x * i;$

$endwhile;$
$\{n * b \leqslant j\}$



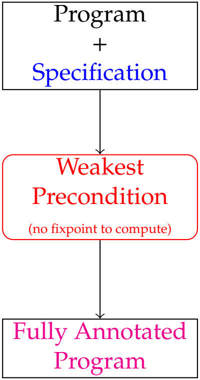
Motivating example

$\{j = 0\}$
$\{j = (b+0) * 0 \wedge b \leqslant (b+0) \wedge 0 \leqslant 0\}$
$i := 0;$
$\{j = (b+i) * i \wedge b \leqslant (b+i) \wedge 0 \leqslant i\}$
$x := b + i;$
$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$while(i! = n)$

$\quad i := c + i$

$\quad j := x * i;$

$endwhile;$
$\{n * b \leqslant j\}$

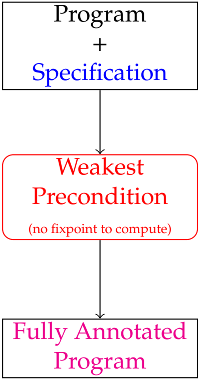


Motivating example

$\{j = 0\}$
$\{j = (b + 0) * 0 \wedge b \leqslant (b + 0) \wedge 0 \leqslant 0\}$
$i := 0;$
$\{j = (b + i) * i \wedge b \leqslant (b + i) \wedge 0 \leqslant i\}$
$x := b + i;$
$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$while(i! = n)$

$\quad i := c + i$

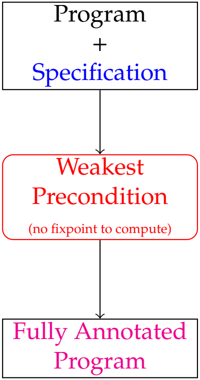
$\quad j := x * i;$
$\{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$endwhile;$
$\{n * b \leqslant j\}$



Motivating example

$\{j = 0\}$
$\{j = (b+0) * 0 \wedge b \leqslant (b+0) \wedge 0 \leqslant 0\}$
$i := 0;$
$\{j = (b+i) * i \wedge b \leqslant (b+i) \wedge 0 \leqslant i\}$
$x := b + i;$
$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$while(i! = n)$

$\quad i := c + i$
$\{x * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$\quad j := x * i;$
$\{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$endwhile;$
$\{n * b \leqslant j\}$



Motivating example

$\{j = 0\}$

$\{j = (b+0) * 0 \wedge b \leqslant (b+0) \wedge 0 \leqslant 0\}$

$i := 0;$

$\{j = (b+i) * i \wedge b \leqslant (b+i) \wedge 0 \leqslant i\}$

$x := b + i;$

$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$while(i! = n)$

$\{x * (c+i) = x * (c+i) \wedge b \leqslant x \wedge 0 \leqslant c + i\}$

$\quad i := c + i$

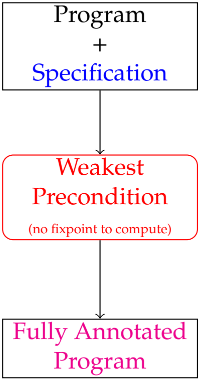
$\{x * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$\quad j := x * i;$

$\{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$endwhile;$

$\{n * b \leqslant j\}$



Motivating example

$\{j = 0\}$
$\{j = (b+0) * 0 \land b \leqslant (b+0) \land 0 \leqslant 0\}$
$i := 0;$
$\{j = (b+i) * i \land b \leqslant (b+i) \land 0 \leqslant i\}$
$x := b + i;$
$\{Inv : j = x * i \land b \leqslant x \land 0 \leqslant i\}$
$while\,(i! = n)$
$\{x * (c+i) = x * (c+i) \land b \leqslant x \land 0 \leqslant c + i\}$
  $i := c + i$
  $j := x * i;$
$endwhile;$
$\{n * b \leqslant j\}$

Set of Proof Obligations:

- $j = 0 \Rightarrow j = (b+0) * 0 \land b \leqslant (b+0) \land 0 \leqslant 0$
- $j = x * i \land b \leqslant x \land 0 \leqslant i \land i \neq n \Rightarrow$
  $x * (c+i) = x * (c+i) \land b \leqslant x \land 0 \leqslant c + i$
- $j = x * i \land b \leqslant x \land 0 \leqslant i \land i = n \Rightarrow n * b \leqslant j$

Constant propagation analysis

$\{j = 0\}$

$\{j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0\}$

$i := 0;$

$\{j = b * i \wedge b \leqslant b \wedge 0 \leqslant i\}$

$(i, 0) \rightarrow \quad x := b + i;$

$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$(x, b) \rightarrow \quad while(i! = n)$

$\{b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i\}$

$(x, b) \rightarrow \quad i := c + i$

$\{b * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$(x, b) \rightarrow \quad j := x * i;$

$\{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$endwhile;$

$\{n * b \leqslant j\}$

Program transformation

$$
\begin{array}{rl}
 & \{j = 0\} \\
 & \{j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0\} \\
 & i := 0; \\
 & \{j = b * i \wedge b \leqslant b \wedge 0 \leqslant i\} \\
(i, 0) \rightarrow & x := b; \\
 & \{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
(x, b) \rightarrow & while(i! = n) \\
 & \{b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i\} \\
(x, b) \rightarrow & \quad i := c + i \\
 & \{b * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
(x, b) \rightarrow & \quad j := x * i; \\
 & \{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
 & endwhile; \\
 & \{n * b \leqslant j\}
\end{array}
$$

Program transformation

$$
\begin{array}{rl}
 & \{j = 0\} \\
 & \{j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0\} \\
 & i := 0; \\
 & \{j = b * i \wedge b \leqslant b \wedge 0 \leqslant i\} \\
(i, 0) \rightarrow & x := b; \\
 & \{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
(x, b) \rightarrow & while(i! = n) \\
 & \{b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i\} \\
(x, b) \rightarrow & \quad i := c + i \\
 & \{b * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
(x, b) \rightarrow & \quad j := b * i; \\
 & \{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
 & endwhile; \\
 & \{n * b \leqslant j\}
\end{array}
$$

WP Computation of optimized program

$\{j = 0\}$

$\{j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0\}$

$i := 0;$

$\{j = b * i \wedge b \leqslant b \wedge 0 \leqslant i\}$

$(i, 0) \rightarrow \quad x := b;$

$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$(x, b) \rightarrow \quad while(i! = n)$

$\{b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i\}$

$(x, b) \rightarrow \quad i := c + i$

$\{b * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$(x, b) \rightarrow \quad j := b * i;$

$\{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$endwhile;$

$\{n * b \leqslant j\}$

WP Computation of optimized program

$\{j = 0\}$

$\{j = b * 0 \land b \leqslant b \land 0 \leqslant 0\}$

$i := 0;$

$\{j = b * i \land b \leqslant b \land 0 \leqslant i\}$

$(i, 0) \rightarrow \quad x := b;$

$\{Inv : j = x * i \land b \leqslant x \land 0 \leqslant i\}$

$(x, b) \rightarrow \quad while(i! = n)$

$\{b * (c + i) = x * (c + i) \land b \leqslant x \land 0 \leqslant c + i\}$

$(x, b) \rightarrow \quad \quad i := c + i$

$\{b * i = x * i \land b \leqslant x \land 0 \leqslant i\}$

$(x, b) \rightarrow \quad \quad j := b * i;$

$\{j = x * i \land b \leqslant x \land 0 \leqslant i\}$

$endwhile;$

$\{n * b \leqslant j\}$

WP Computation of optimized program

$$
\begin{array}{ll}
 & \{j = 0\} \\
 & \{j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0\} \\
 & i := 0; \\
 & \{j = b * i \wedge b \leqslant b \wedge 0 \leqslant i\} \\
(i, 0) \rightarrow & x := b; \\
 & \{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
(x, b) \rightarrow & while(i! = n) \\
 & \{b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i\} \\
(x, b) \rightarrow & \quad i := c + i \\
 & \{b * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
(x, b) \rightarrow & \quad j := b * i; \\
 & \{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
 & endwhile; \\
 & \{n * b \leqslant j\}
\end{array}
$$

WP Computation of optimized program

$\{j = 0\}$

$\{j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0\}$

$i := 0;$

$\{j = b * i \wedge b \leqslant b \wedge 0 \leqslant i\}$

$(i, 0) \to \quad x := b;$

$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$(x, b) \to \quad while(i! = n)$

$\{b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i\}$

$(x, b) \to \quad i := c + i$

$\{b * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$(x, b) \to \quad j := b * i;$

$\{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$endwhile;$

$\{n * b \leqslant j\}$

WP Computation of optimized program

$$
\begin{array}{rl}
 & \{j = 0\} \\
 & \{j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0\} \\
 & i := 0; \\
 & \{j = b * i \wedge b \leqslant b \wedge 0 \leqslant i\} \\
(i, 0) \rightarrow & x := b; \\
 & \{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
(x, b) \rightarrow & while(i! = n) \\
 & \{b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i\} \\
(x, b) \rightarrow & \quad i := c + i \\
 & \{b * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
(x, b) \rightarrow & \quad j := b * i; \\
 & \{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\} \\
 & endwhile; \\
 & \{n * b \leqslant j\}
\end{array}
$$

Proof Obligations

$\{j = 0\}$
$\{j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0\}$
$i := 0;$
$\{j = b * i \wedge b \leqslant b \wedge 0 \leqslant i\}$
$x := b;$
$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$while\,(i! = n)$
$\{b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i\}$
$\quad i := c + i$
$\{b * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$\quad j := b * i;$
$\{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$endwhile;$
$\{n * b \leqslant j\}$

Proof Obligations:

1. $j = 0 \Rightarrow j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0$
2. $j = x * i \wedge b \leqslant x \wedge 0 \leqslant i \wedge i \neq n$
   $\Rightarrow b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i$
3. $j = x * i \wedge b \leqslant x \wedge 0 \leqslant i \wedge i = n \Rightarrow n * b \leqslant j$

Proof Obligations

$\{j = 0\}$
$\{j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0\}$
$i := 0;$
$\{j = b * i \wedge b \leqslant b \wedge 0 \leqslant i\}$
$x := b;$
$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$while\,(\,i! = n\,)$
$\{b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i\}$
$\quad i := c + i$
$\{b * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$\quad j := b * i;$
$\{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$endwhile;$
$\{n * b \leqslant j\}$

Proof Obligations:

1. $j = 0 \Rightarrow j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0$
2. $j = x * i \wedge b \leqslant x \wedge 0 \leqslant i \wedge i \neq n \Rightarrow b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i$ — Unprovable without knowing $x = b$
3. $j = x * i \wedge b \leqslant x \wedge 0 \leqslant i \wedge i = n \Rightarrow n * b \leqslant j$

Proof Obligations

$\{j = 0\}$
$\{j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0\}$
$i := 0;$
$\{j = b * i \wedge b \leqslant b \wedge 0 \leqslant i\}$
$x := b;$
$\{Inv : j = x * i \wedge b \leqslant x \wedge 0 \leqslant i \wedge x = b\}$
while $(i != n)$
$\{b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i\}$
$\quad i := c + i$
$\{b * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$\quad j := b * i;$
$\{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
endwhile;
$\{n * b \leqslant j\}$

Proof Obligations:

1. $j = 0 \Rightarrow j = b * 0 \wedge b \leqslant b \wedge 0 \leqslant 0$
2. $j = x * i \wedge b \leqslant x \wedge 0 \leqslant i \wedge x = b \wedge i \neq n$
   $\Rightarrow b * (c + i) = x * (c + i) \wedge b \leqslant x \wedge 0 \leqslant c + i$
3. $j = x * i \wedge b \leqslant x \wedge 0 \leqslant i \wedge i = n \Rightarrow n * b \leqslant j$

Solution: strengthen annotations

Strengthening annotations

- allows to verify proof obligations of original program
- but also introduces new proof obligations

S_1		S_1
$\{\varphi_1\}$		$\{\varphi_1 \wedge \psi_1\}$
S_2		S_2
$\{\varphi_2\}$	$\rightsquigarrow$	$\{\varphi_2 \wedge \psi_2\}$
S_3		S_3
$\{\varphi_3\}$		$\{\varphi_3 \wedge \psi_3\}$

- $\varphi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2)$
- $\varphi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3)$

- $\varphi_1 \wedge \psi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2 \wedge \psi_2)$
- $\varphi_2 \wedge \psi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3 \wedge \psi_3)$

If the analysis is correct,

- $\psi_1 \Rightarrow \mathsf{wp}(S_1, \psi_2)$
- $\psi_2 \Rightarrow \mathsf{wp}(S_2, \psi_3)$

are valid proof obligations.

Strengthening annotations

- allows to verify proof obligations of original program
- but also introduces new proof obligations

$$\begin{array}{lcl} S_1 & & S_1 \\ \{\varphi_1\} & & \{\varphi_1 \wedge \psi_1\} \\ S_2 & & S_2 \\ \{\varphi_2\} & \rightsquigarrow & \{\varphi_2 \wedge \psi_2\} \\ S_3 & & S_3 \\ \{\varphi_3\} & & \{\varphi_3 \wedge \psi_3\} \end{array}$$

- $\varphi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2)$
- $\varphi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3)$

- $\varphi_1 \wedge \psi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2 \wedge \psi_2)$
- $\varphi_2 \wedge \psi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3 \wedge \psi_3)$

If the analysis is correct,

- $\psi_1 \Rightarrow \mathsf{wp}(S_1, \psi_2)$
- $\psi_2 \Rightarrow \mathsf{wp}(S_2, \psi_3)$

are valid proof obligations.

Strengthening annotations

- allows to verify proof obligations of original program
- but also introduces new proof obligations

$$\begin{array}{lcl} S_1 & & S_1 \\ \{\varphi_1\} & & \{\varphi_1 \wedge \psi_1\} \\ S_2 & & S_2 \\ \{\varphi_2\} & \rightsquigarrow & \{\varphi_2 \wedge \psi_2\} \\ S_3 & & S_3 \\ \{\varphi_3\} & & \{\varphi_3 \wedge \psi_3\} \end{array}$$

- $\varphi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2)$
- $\varphi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3)$

- $\varphi_1 \wedge \psi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2 \wedge \psi_2)$
- $\varphi_2 \wedge \psi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3 \wedge \psi_3)$

If the analysis is correct,

- $\psi_1 \Rightarrow \mathsf{wp}(S_1, \psi_2)$
- $\psi_2 \Rightarrow \mathsf{wp}(S_2, \psi_3)$

are valid proof obligations.

Strengthening annotations

- allows to verify proof obligations of original program
- but also introduces new proof obligations

$$S_1 \; \{\varphi_1\} \; S_2 \; \{\varphi_2\} \; S_3 \; \{\varphi_3\} \quad \rightsquigarrow \quad S_1 \; \{\varphi_1 \wedge \psi_1\} \; S_2 \; \{\varphi_2 \wedge \psi_2\} \; S_3 \; \{\varphi_3 \wedge \psi_3\}$$

- $\varphi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2)$
- $\varphi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3)$

- $\varphi_1 \wedge \psi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2 \wedge \psi_2)$
- $\varphi_2 \wedge \psi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3 \wedge \psi_3)$

If the analysis is correct,

- $\psi_1 \Rightarrow \mathsf{wp}(S_1, \psi_2)$
- $\psi_2 \Rightarrow \mathsf{wp}(S_2, \psi_3)$

are valid proof obligations.

Strengthening annotations

- allows to verify proof obligations of original program
- but also introduces new proof obligations

$$
\begin{array}{l} S_1 \\ \{\varphi_1\} \\ S_2 \\ \{\varphi_2\} \\ S_3 \\ \{\varphi_3\} \end{array}
\quad \rightsquigarrow \quad
\begin{array}{l} S_1 \\ \{\varphi_1 \wedge \psi_1\} \\ S_2 \\ \{\varphi_2 \wedge \psi_2\} \\ S_3 \\ \{\varphi_3 \wedge \psi_3\} \end{array}
$$

- $\varphi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2)$
- $\varphi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3)$

- $\varphi_1 \wedge \psi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2) \wedge \mathsf{wp}(S_1, \psi_2)$
- $\varphi_2 \wedge \psi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3) \wedge \mathsf{wp}(S_2, \psi_3)$

If the analysis is correct,

- $\psi_1 \Rightarrow \mathsf{wp}(S_1, \psi_2)$
- $\psi_2 \Rightarrow \mathsf{wp}(S_2, \psi_3)$

are valid proof obligations.

Strengthening annotations

- allows to verify proof obligations of original program
- but also introduces new proof obligations

$$\begin{array}{l} S_1 \\ \{\varphi_1\} \\ S_2 \\ \{\varphi_2\} \\ S_3 \\ \{\varphi_3\} \end{array} \quad \rightsquigarrow \quad \begin{array}{l} S_1 \\ \{\varphi_1 \wedge \psi_1\} \\ S_2 \\ \{\varphi_2 \wedge \psi_2\} \\ S_3 \\ \{\varphi_3 \wedge \psi_3\} \end{array}$$

- $\varphi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2)$
- $\varphi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3)$

- $\varphi_1 \wedge \psi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2) \wedge \mathsf{wp}(S_1, \psi_2)$
- $\varphi_2 \wedge \psi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3) \wedge \mathsf{wp}(S_2, \psi_3)$

If the analysis is correct,

- $\psi_1 \Rightarrow \mathsf{wp}(S_1, \psi_2)$
- $\psi_2 \Rightarrow \mathsf{wp}(S_2, \psi_3)$

are valid proof obligations.

Certifying/Proof producing analyzer

A certifying analyzer extends a standard analyzer with a procedure that generates a certificate for the result of the analysis

- Certifying analyzers exist under mild hypotheses:
  - results of the analysis expressible as assertions
  - abstract transfer functions are correct w.r.t. wp
  - ...
- Ad hoc construction of certificates yields compact certificates

Certifying/Proof producing analyzer

A certifying analyzer extends a standard analyzer with a procedure that generates a certificate for the result of the analysis

- Certifying analyzers exist under mild hypotheses:
  - results of the analysis expressible as assertions
  - abstract transfer functions are correct w.r.t. wp
  - ...
- Ad hoc construction of certificates yields compact certificates

Certifying/Proof producing analyzer

A certifying analyzer extends a standard analyzer with a procedure that generates a certificate for the result of the analysis

- Certifying analyzers exist under mild hypotheses:
  - results of the analysis expressible as assertions
  - abstract transfer functions are correct w.r.t. wp
  - ...
- Ad hoc construction of certificates yields compact certificates

Certifying analysis for constant propagation

$\{true\}$
$\{b = b\}$
$i := 0;$
$\{b = b\}$
$x := b;$
$\{Inv : x = b\}$
$while(i! = n)$
$\{x = b\}$
$\quad i := c + i$
$\{x = b\}$
$\quad j := b * i;$
$\{x = b\}$
$endwhile;$
$\{true\}$

Certifying analysis for constant propagation

$\{true\}$
$\{b = b\}$
$i := 0;$
$\{b = b\}$
$x := b;$
$\{Inv : x = b\}$
$while(i! = n)$
$\{x = b\}$
$\quad i := c + i$
$\{x = b\}$
$\quad j := b * i;$
$\{x = b\}$
$endwhile;$
$\{true\}$

With proof obligations:
$x = b \wedge i = n \Rightarrow true$
$x = b \wedge i \neq n \Rightarrow x = b$
$true \Rightarrow b = b$

$$
\begin{array}{ccccccc}
\{\phi_1\} & + & \{\phi_1^A\} & \longrightarrow & \{\phi_1 \wedge \phi_1^A\} & & \\
S_1 & & S_1 & & S_1 & \longrightarrow & S_1^O \\
\{\phi_2\} & + & \{\phi_2^A\} & \longrightarrow & \{\phi_2 \wedge \phi_2^A\} & & \\
S_2 & & S_2 & & S_2 & \longrightarrow & S_2^O \\
\vdots & + & \vdots & \longrightarrow & \vdots & & \vdots \\
S_{n-1} & & S_{n-1} & & S_{n-1} & \longrightarrow & S_{n-1}^O \\
\{\phi_n\} & + & \{\phi_n^A\} & \longrightarrow & \{\phi_n \wedge \phi_n^A\} & & \\
S_n & & S_n & & S_n & \longrightarrow & S_n^O
\end{array}
$$

Translation consists of:

1. Specifying and certifying automatically the result of the analysis
2. Merging annotations (trivial)
3. Merging certificates

$$
\begin{array}{ccccccc}
\{\phi_1\} & + & \{\phi_1^A\} & \rightarrow & \{\phi_1 \wedge \phi_1^A\} & & \{\phi_1' \wedge \phi_1'^A\} \\
S_1 & & S_1 & & S_1 & \longrightarrow & S_1^O \\
\{\phi_2\} & + & \{\phi_2^A\} & \rightarrow & \{\phi_2 \wedge \phi_2^A\} & & \{\phi_2' \wedge \phi_2'^A\} \\
S_2 & & S_2 & & S_2 & \longrightarrow & S_2^O \\
\vdots & + & \vdots & \rightarrow & \vdots & & \vdots \\
S_{n-1} & & S_{n-1} & & S_{n-1} & \longrightarrow & S_{n-1}^O \\
\{\phi_n\} & + & \{\phi_n^A\} & \rightarrow & \{\phi_n \wedge \phi_n^A\} & & \{\phi_n' \wedge \phi_n'^A\} \\
S_n & & S_n & & S_n & \longrightarrow & S_n^O
\end{array}
$$

Translation consists of:

1. Specifying and certifying automatically the result of the analysis
2. Merging annotations (trivial)
3. Merging certificates

$$
\begin{array}{ccccccc}
\{\phi_1\} & + & \{\phi_1^A\} & \rightarrow & \{\phi_1 \wedge \phi_1^A\} & & \{\phi_1' \wedge \phi_1^A\} \\
S_1 & & S_1 & & S_1 & \rightarrow & S_1^O \\
\{\phi_2\} & + & \{\phi_2^A\} & \rightarrow & \{\phi_2 \wedge \phi_2^A\} & & \{\phi_2' \wedge \phi_2^A\} \\
S_2 & & S_2 & & S_2 & \rightarrow & S_2^O \\
\vdots & + & \vdots & \rightarrow & \vdots & & \vdots \\
S_{n-1} & & S_{n-1} & & S_{n-1} & \rightarrow & S_{n-1}^O \\
\{\phi_n\} & + & \{\phi_n^A\} & \rightarrow & \{\phi_n \wedge \phi_n^A\} & & \{\phi_n' \wedge \phi_n^A\} \\
S_n & & S_n & & S_n & \rightarrow & S_n^O
\end{array}
$$

Translation consists of:

1. Specifying and certifying automatically the result of the analysis
2. Merging annotations (trivial)
3. Merging certificates

$$
\begin{array}{ccccccc}
\{\phi_1\} & + & \{\phi_1^A\} & \rightarrow & \{\phi_1 \wedge \phi_1^A\} & & \{\phi_1' \wedge \phi_1^A\} \\
S_1 & & S_1 & & S_1 & \rightarrow & S_1^O \\
\{\phi_2\} & + & \{\phi_2^A\} & \rightarrow & \{\phi_2 \wedge \phi_2^A\} & & \{\phi_2' \wedge \phi_2^A\} \\
S_2 & & S_2 & & S_2 & \rightarrow & S_2^O \\
\vdots & + & \vdots & \rightarrow & \vdots & & \vdots \\
S_{n-1} & & S_{n-1} & & S_{n-1} & \rightarrow & S_{n-1}^O \\
\{\phi_n\} & + & \{\phi_n^A\} & \rightarrow & \{\phi_n \wedge \phi_n^A\} & & \{\phi_n' \wedge \phi_n^A\} \\
S_n & & S_n & & S_n & \rightarrow & S_n^O
\end{array}
$$

Translation consists of:

1. Specifying and certifying automatically the result of the analysis
2. Merging annotations (trivial)
3. Merging certificates

Certificates

Merging of certificates is not tied to a particular certificate format, but to the existence of functions to manipulate them.

Proof algebra

$$
\begin{array}{lll}
\textsf{axiom} & : & \mathcal{P}(\Gamma; A; \Delta \vdash A) \\
\textsf{ring} & : & \mathcal{P}(\Gamma \vdash n_1 = n_2) \qquad \text{if } n_1 = n_2 \text{ is a ring equality} \\
\textsf{intro}_{\Rightarrow} & : & \mathcal{P}(\Gamma; A \vdash B) \to \mathcal{P}(\Gamma \vdash A \Rightarrow B) \\
\textsf{elim}_{\Rightarrow} & : & \mathcal{P}(\Gamma \vdash A \Rightarrow B) \to \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash B) \\
\textsf{elim}_{=} & : & \mathcal{P}(\Gamma \vdash e_1 = e_2) \to \mathcal{P}(\Gamma \vdash A[{}^{e_1}\!/_r]) \to \mathcal{P}(\Gamma \vdash A[{}^{e_2}\!/_r]) \\
\textsf{subst} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma[{}^{e}\!/_r] \vdash A[{}^{e}\!/_r])
\end{array}
$$

Merging certificates

We need to build from the original and analysis certificates:

$$\frac{\phi_1 \Rightarrow \mathsf{wp}(S, \phi_2)}{\{\phi_1\}S\{\phi_2\}} \qquad \frac{a_1 \Rightarrow \mathsf{wp}(S, a_2)}{\{a_1\}S\{a_2\}}$$

the certificate for the optimized program:

$$\frac{\phi_1 \wedge a_1 \Rightarrow \mathsf{wp}(S', \phi_2 \wedge a_2)}{\{\phi_1 \wedge a_1\}S'\{\phi_2 \wedge a_2\}}$$

by using the gluing lemma

$$\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$$

where ins' is the optimization of ins, and a is the result of the analysis

We really construct by well-founded induction a proof term of

$$\mathsf{wp}_P(k) \wedge a(k) \implies \mathsf{wp}_{P'}(k)$$

Merging certificates

We need to build from the original and analysis certificates:

$$\frac{\phi_1 \Rightarrow \mathsf{wp}(S, \phi_2)}{\{\phi_1\}S\{\phi_2\}} \qquad \frac{a_1 \Rightarrow \mathsf{wp}(S, a_2)}{\{a_1\}S\{a_2\}}$$

the certificate for the optimized program:

$$\frac{\phi_1 \wedge a_1 \Rightarrow \mathsf{wp}(S', \phi_2 \wedge a_2)}{\{\phi_1 \wedge a_1\}S'\{\phi_2 \wedge a_2\}}$$

by using the gluing lemma

$$\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$$

where ins' is the optimization of ins, and a is the result of the analysis

We really construct by well-founded induction a proof term of

$$\mathsf{wp}_P(k) \wedge a(k) \implies \mathsf{wp}_{P'}(k)$$

Merging certificates

We need to build from the original and analysis certificates:

$$\frac{\phi_1 \Rightarrow \mathsf{wp}(S, \phi_2)}{\{\phi_1\}S\{\phi_2\}} \qquad \frac{a_1 \Rightarrow \mathsf{wp}(S, a_2)}{\{a_1\}S\{a_2\}}$$

the certificate for the optimized program:

$$\frac{\phi_1 \wedge a_1 \Rightarrow \mathsf{wp}(S', \phi_2 \wedge a_2)}{\{\phi_1 \wedge a_1\}S'\{\phi_2 \wedge a_2\}}$$

by using the gluing lemma

$$\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$$

where ins' is the optimization of ins, and a is the result of the analysis

We really construct by well-founded induction a proof term of

$$\mathsf{wp}_P(k) \wedge a(k) \implies \mathsf{wp}_{P'}(k)$$

Merging certificates

We need to build from the original and analysis certificates:

$$\frac{\phi_1 \Rightarrow \mathsf{wp}(S, \phi_2)}{\{\phi_1\}S\{\phi_2\}} \qquad \frac{a_1 \Rightarrow \mathsf{wp}(S, a_2)}{\{a_1\}S\{a_2\}}$$

the certificate for the optimized program:

$$\frac{\phi_1 \wedge a_1 \Rightarrow \mathsf{wp}(S', \phi_2 \wedge a_2)}{\{\phi_1 \wedge a_1\}S'\{\phi_2 \wedge a_2\}}$$

by using the gluing lemma

$$\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$$

where ins' is the optimization of ins, and a is the result of the analysis

We really construct by well-founded induction a proof term of

$$\mathsf{wp}_P(k) \wedge a(k) \implies \mathsf{wp}_{P'}(k)$$

Illustrating: $\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$

If the value of e is known to be n, then

$$\begin{array}{ccc} \dots & & \dots \\ y := e & \xrightarrow{n=e} & y := n \\ \dots & & \dots \end{array}$$

The gluing lemma states in this case:

Under the hypothesis that the result of the analysis is valid $n = e$ the weakest precondition applied to the transformed instruction

$$\mathsf{wp}(y := n, \varphi) \quad (\equiv \varphi[^n/_y])$$

can be derived from the original one:

$$\mathsf{wp}(y := e, \varphi) \quad (\equiv \varphi[^e/_y])$$

Illustrating: $\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$

If the value of e is known to be n, then

$$\begin{array}{lcl} \ldots & & \ldots \\ y := e & \xrightarrow{n=e} & y := n \\ \ldots & & \ldots \end{array}$$

The gluing lemma states in this case:

Under the hypothesis that the result of the analysis is valid $n = e$ the weakest precondition applied to the transformed instruction

$$\mathsf{wp}(y := n, \phi) \quad (= \phi[^n/_y])$$

can be derived from the original one:

$$\mathsf{wp}(y := e, \phi) \quad (= \phi[^e/_y])$$

Illustrating: $\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$

If the value of e is known to be n, then

$$\begin{array}{l} \ldots \\ y := e \\ \ldots \end{array} \quad \xrightarrow{n=e} \quad \begin{array}{l} \ldots \\ y := n \\ \ldots \end{array}$$

The gluing lemma states in this case:

Under the hypothesis that the result of the analysis is valid $n = e$ the weakest precondition applied to the transformed instruction

$$\mathsf{wp}(y := n, \varphi) \quad (= \varphi[^n/_y])$$

can be derived from the original one:

$$\mathsf{wp}(y := e, \varphi) \quad (= \varphi[^e/_y])$$

Illustrating: $\forall\phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$

If the value of e is known to be n, then

$$\begin{array}{lcl} \ldots & & \ldots \\ y := e & \xrightarrow{n=e} & y := n \\ \ldots & & \ldots \end{array}$$

The gluing lemma states in this case:

Under the hypothesis that the result of the analysis is valid $n = e$

the weakest precondition applied to the transformed instruction

$$\mathsf{wp}(y := n, \varphi) \quad (\equiv \varphi[^n/_y])$$

can be derived from the original one:

$$\mathsf{wp}(y := e, \varphi) \quad (\equiv \varphi[^e/_y])$$

Illustrating: $\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$

If the value of e is known to be n, then

$$\begin{array}{l} \dots \\ y := e \\ \dots \end{array} \quad \xrightarrow{n=e} \quad \begin{array}{l} \dots \\ y := n \\ \dots \end{array}$$

The gluing lemma states in this case:

Under the hypothesis that the result of the analysis is valid $n = e$

the weakest precondition applied to the transformed instruction

$$\mathsf{wp}(y := n, \varphi) \quad (\equiv \varphi[{}^{n}/_{y}])$$

can be derived from the original one:

$$\mathsf{wp}(y := e, \varphi) \quad (\equiv \varphi[{}^{e}/_{y}])$$

Illustrating: $\forall\phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$

If the value of e is known to be n, then

$$\begin{array}{lcl} \ldots & & \ldots \\ y := e & \xrightarrow{n=e} & y := n \\ \ldots & & \ldots \end{array}$$

The gluing lemma states in this case:

Under the hypothesis that the result of the analysis is valid $n = e$

the weakest precondition applied to the transformed instruction

$$\mathsf{wp}(y := n, \varphi) \quad (\equiv \varphi[{}^{n}\!/\!_{y}])$$

can be derived from the original one:

$$\mathsf{wp}(y := e, \varphi) \quad (\equiv \varphi[{}^{e}\!/\!_{y}])$$

Illustrating: $\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$

$\{\varphi_1\}$
$x := 5;$
$\{\varphi_2\}$
$y := x$
$\{\varphi_3\}$

$\{T\}$
$x := 5;$
$\{x = 5\}$
$y := x$
$\{x = 5\}$

$\{\varphi_1 \wedge T\}$
$x := 5;$
$\{\varphi_2 \wedge x = 5\}$
$y := 5$
$\{\varphi_3 \wedge x = 5\}$

Original PO's:

- $\varphi_1 \Rightarrow \varphi_2[{}^{5}\!/_{x}]$
- $\varphi_2 \Rightarrow \varphi_3[{}^{x}\!/_{y}]$

Analysis PO's :

- $T \Rightarrow 5 = 5$
- $x = 5 \Rightarrow x = 5$

Final PO's:

- $\varphi_1 \wedge T \Rightarrow \varphi_2[{}^{5}\!/_{x}] \wedge 5 = 5$
- $\varphi_2 \wedge x = 5 \Rightarrow \varphi_3[{}^{5}\!/_{y}] \wedge x = 5$

Illustrating: $\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$

$\{\varphi_1\}$
$x := 5;$
$\{\varphi_2\}$
$y := x$
$\{\varphi_3\}$

$\{T\}$
$x := 5;$
$\{x = 5\}$
$y := x$
$\{x = 5\}$

$\{\varphi_1 \wedge T\}$
$x := 5;$
$\{\varphi_2 \wedge x = 5\}$
$y := 5$
$\{\varphi_3 \wedge x = 5\}$

Original PO's:

- $\varphi_1 \Rightarrow \varphi_2[5/x]$
- $\varphi_2 \Rightarrow \varphi_3[x/y]$

Analysis PO's :

- $T \Rightarrow 5 = 5$
- $x = 5 \Rightarrow x = 5$

Final PO's:

- $\varphi_1 \wedge T \Rightarrow \varphi_2[5/x] \wedge 5 = 5$
- $\varphi_2 \wedge x = 5 \Rightarrow \varphi_3[5/y] \wedge x = 5$

Illustrating: $\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$

$\{\varphi_1\}$
$x := 5;$
$\{\varphi_2\}$
$y := x$
$\{\varphi_3\}$

$\{T\}$
$x := 5;$
$\{x = 5\}$
$y := x$
$\{x = 5\}$

$\{\varphi_1 \wedge T\}$
$x := 5;$
$\{\varphi_2 \wedge x = 5\}$
$y := 5$
$\{\varphi_3 \wedge x = 5\}$

Original PO's:

- $\varphi_1 \Rightarrow \varphi_2[5/x]$
- $\varphi_2 \Rightarrow \varphi_3[x/y]$

Analysis PO's :

- $T \Rightarrow 5 = 5$
- $x = 5 \Rightarrow x = 5$

Final PO's:

- $\varphi_1 \wedge T \Rightarrow \varphi_2[5/x] \wedge 5 = 5$
- $\varphi_2 \wedge x = 5 \Rightarrow \varphi_3[5/y] \wedge x = 5$

Illustrating: $\forall \phi, \mathsf{wp}(\mathsf{ins}, \phi) \wedge a \Rightarrow \mathsf{wp}(\mathsf{ins}', \phi)$

$\{\varphi_1\}$
$x := 5;$
$\{\varphi_2\}$
$y := x$
$\{\varphi_3\}$

$\{T\}$
$x := 5;$
$\{x = 5\}$
$y := x$
$\{x = 5\}$

$\{\varphi_1 \wedge T\}$
$x := 5;$
$\{\varphi_2 \wedge x = 5\}$
$y := 5$
$\{\varphi_3 \wedge x = 5\}$

Original PO's:

- $\varphi_1 \Rightarrow \varphi_2[^5/_x]$
- $\varphi_2 \Rightarrow \varphi_3[^x/_y]$

Analysis PO's :

- $T \Rightarrow 5 = 5$
- $x = 5 \Rightarrow x = 5$

Final PO's:

- $\varphi_1 \wedge T \Rightarrow \varphi_2[^5/_x] \wedge 5 = 5$
- $\varphi_2 \wedge x = 5 \Rightarrow \varphi_3[^5/_y] \wedge x = 5$

Applicability and justification of method

Certificate translation is applicable to many common program optimizations:

- Constant propagation
- Loop induction register strength reduction
- Common subexpression elimination
- Dead register elimination
- Register allocation
- Inlining
- Dead code elimination

However,

- particular language
- particular VCgen
- particular program optimizations

} provide a general and unifying framework

An Abstract Model for Certificate Translation

1. We use abstract interpretation to capture in a single model
   - interactive verification
   - automatic program analysis
2. We provide sufficient conditions for existence of certifying analyzers and certificate translators

Abstract interpretation is a natural framework to achieve crisp formalizations of certificate translation

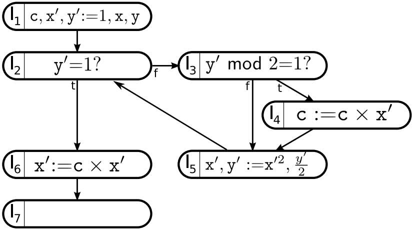
Benefits of generalization

- Language independent and generic in analysis/verification framework
- Applicable to backwards and forward verification methods
- Extensible

In the sequel, we only consider the case of forward analysis and verification

Program Representation

$c := 1$
$x' := x$
$y' := y$
$\textit{while } (y' \neq 1) \textit{ do}$
$\quad \textit{if } (y' \textit{ mod } 2 = 1) \textit{ then}$
$\quad\quad c := c \times x'$
$\quad \textit{fi}$
$\textit{done}$
$x' = x' \times c$

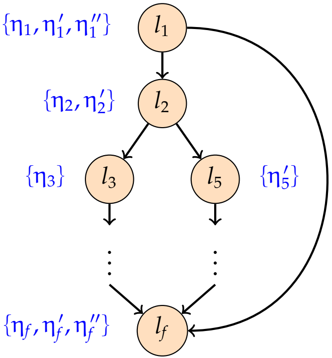


Program: directed graph

- Nodes denoting execution points ($\mathcal{N}$).
- Edges denoting possible transitions between nodes ($\mathcal{E}$).

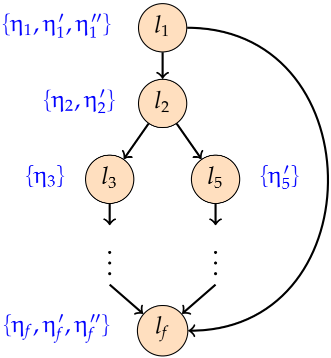
Abstract Interpretation

Program semantics

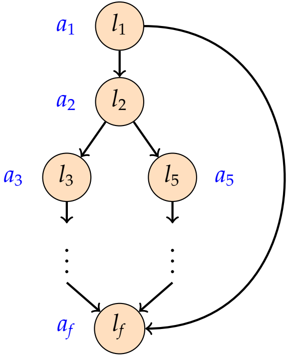


Abstract Interpretation

Program semantics

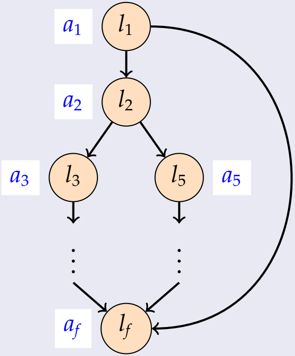


Abstract representation



Solution of a Forward Abstract Interpretation

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

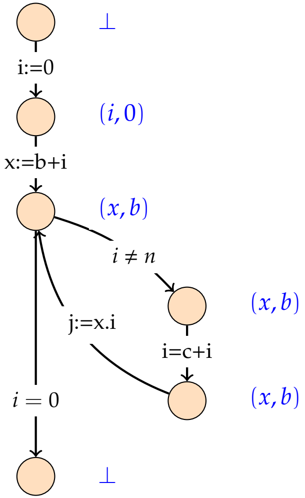


$\{a_1, a_2, \ldots, a_f\}$ a solution of $(\mathbf{D}, T)$ if:

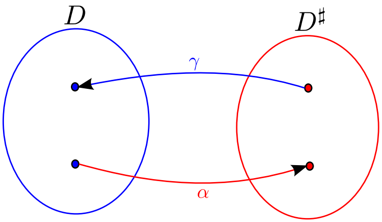
$$
\begin{array}{l}
T_{\langle l_1, l_2 \rangle}(a_1) \sqsubseteq a_2 \\
T_{\langle l_2, l_5 \rangle}(a_2) \sqsubseteq a_5 \\
T_{\langle l_1, l_f \rangle}(a_1) \sqsubseteq a_f \\
\ldots
\end{array}
$$

Example of decidable solution

(D, T): constant analysis (for constant propagation)



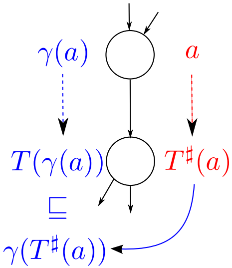
Galois connections capture notion of imprecision



In the following (intuition):

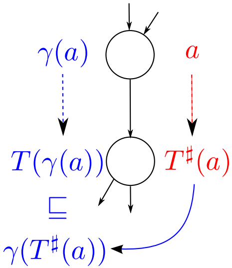
- (D, T): verification framework based on symbolic execution
- $(D^\sharp, T^\sharp)$: static analysis that *justifies* a program optimization.

Consistency of $T^\sharp$ w.r.t. T



$$T(\gamma(a)) \sqsubseteq \gamma(T^\sharp(a))$$

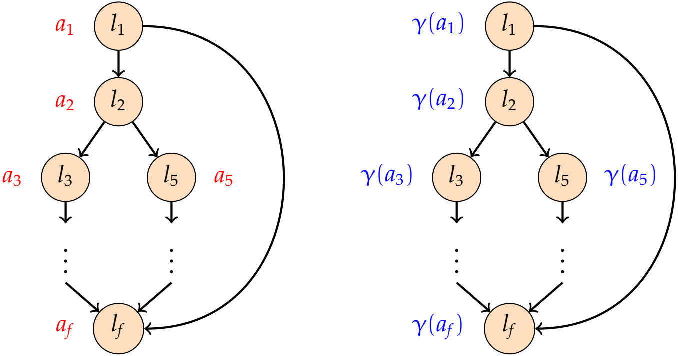
Consistency of $T^\sharp$ w.r.t. T



$$T(\gamma(a)) \sqsubseteq \gamma(T^\sharp(a))$$

Smaller elements: more information

Consistency of $T^\sharp$ w.r.t. T



Result:

$\{a_1, a_2 \ldots a_n\}$ a solution of $(D^\sharp, T^\sharp)$, then $\{\gamma(a_1), \gamma(a_2) \ldots \gamma(a_n)\}$ is a solution of (D, T).

Certified Solutions

Definition

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

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

if $\langle\{a_1 \ldots a_n\}, c_a\rangle$ and $\langle\{b_1 \ldots b_n\}, c_b\rangle$ are certified solutions of D, then $\langle\{a_1 \sqcap b_1 \ldots a_n \sqcap b_n\}, c_a \oplus c_b\rangle$ is a certified solution.

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

Certified Solutions

Definition

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

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

if $(\{a_1 \ldots a_n\}, c_a)$ and $(\{b_1 \ldots b_n\}, c_b)$ are certified solutions of D, then $(\{a_1 \sqcap b_1 \ldots a_n \sqcap b_n\}, c_a \oplus c_b)$ is a certified solution.

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].

Certified Solutions

Definition

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

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

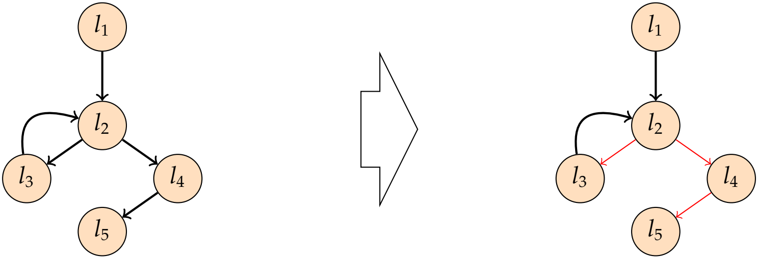
if $(\{a_1 \ldots a_n\}, c_a)$ and $(\{b_1 \ldots b_n\}, c_b)$ are certified solutions of D, then $(\{a_1 \sqcap b_1 \ldots a_n \sqcap b_n\}, c_a \oplus c_b)$ is a certified solution.

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

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

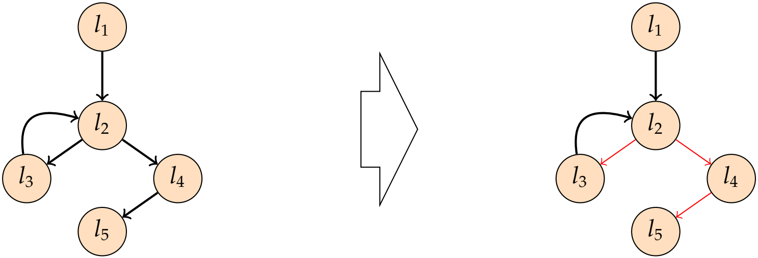
then $(\{\gamma(a_1) \ldots \gamma(a_n)\}, c)$ is a certified solution of (D, T) [for some c].

Program Transformation



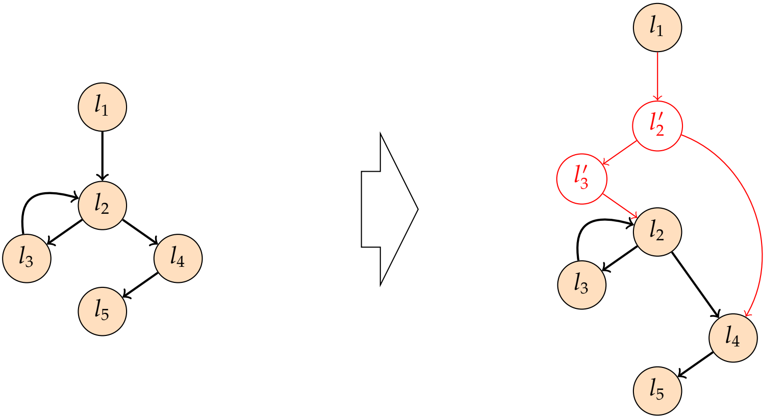
- $T_e \mapsto T'_e, e \in \mathcal{E}$
- a proof of $T'_{\langle l_2, l_3 \rangle}(\_) \sqsubseteq a_3 \sqcap T_{\langle l_2, l_3 \rangle}(\_)$
- const and copy propag / loop induction var strength reduction / common. subexpr elimination / etc.

Program Transformation



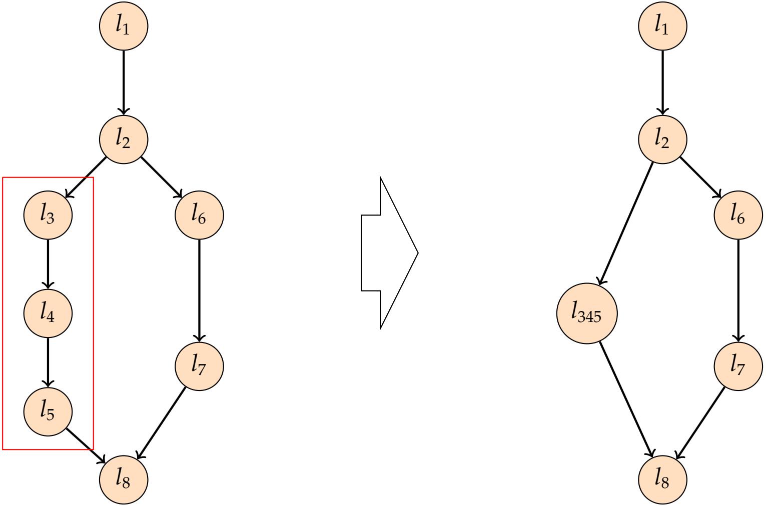
- $T_e \mapsto T'_e, e \in \mathcal{E}$
- a proof of $T'_{\langle l_2,l_3\rangle}(\_) \sqsubseteq a_3 \sqcap T_{\langle l_2,l_3\rangle}(\_)$
- const and copy propag / loop induction var strength reduction / common. subexpr elimination / etc.

Code Duplication



- loop unrolling / function inlining

Node Coalescing



Extensions and prototypes

- We have developed a prototype implementation of a certificate translator.
  - We use ad-hoc methods for certifying analyzers and for transforming certificates along constant propagation/common subexpression elimination.
- Extensions
  - Concurrent and parallel languages
  - Domain-specific languages

Conclusions

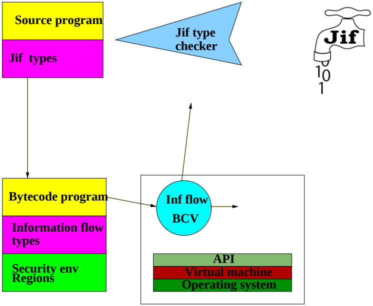
Two verification methods for bytecode and their relation to verification methods for source code

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

Conclusions

Two verification methods for bytecode and their relation to verification methods for source code

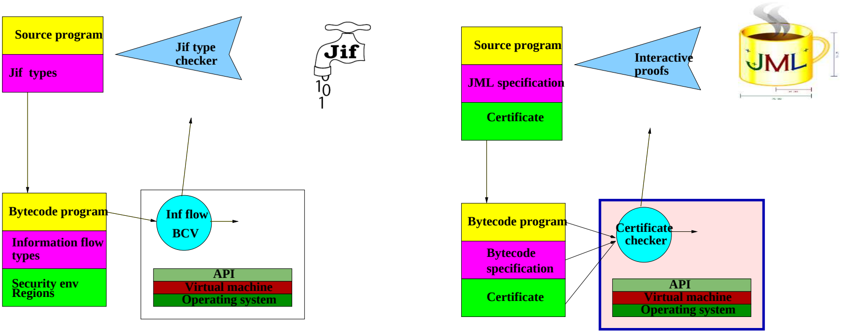
- Type system for information flow based confidentiality policies
- Verification condition generator for logical specifications



Conclusions

Two verification methods for bytecode and their relation to verification methods for source code

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



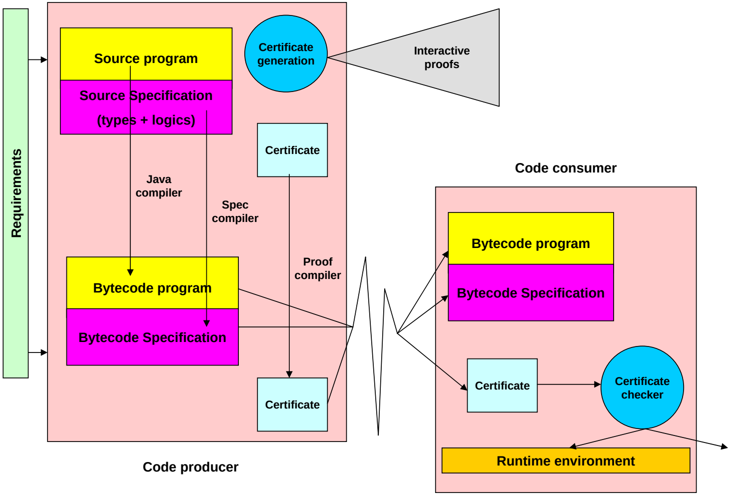
Deployment of secure mobile code can benefit from:

- advanced verification mechanisms at bytecode level
- methods to “compile” evidence from producer to consumer
- machine checked proofs of verification mechanisms on consumer side (use reflection)

Mobius project

- Certified PCC
  - Machine checked certificate checkers
- Basic technologies (type systems and logics) for static enforcement of expressive policies at application level
  - information flow: public outputs should not depends on confidential data
  - resource usage: memory usage, billable actions,...
  - functional correctness: proof-transforming compilation
- Certificate generation by type-preserving compilation, certifying compilation, and proof-transforming compilation
- see `http://mobius.inria.fr`

Mobius view



Requirements
Source program
Source Specification
(types + logics)
Certificate generation
Interactive proofs
Certificate
Java compiler
Spec compiler
Proof compiler
Bytecode program
Bytecode Specification
Certificate
Code producer
Code consumer
Bytecode program
Bytecode Specification
Certificate
Certificate checker
Runtime environment

Further information

Mobius

http://mobius.inria.fr