Language-based methods for software security

Gilles Barthe

IMDEA Software, Madrid, Spain

Part 2

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 $\sim_L$.

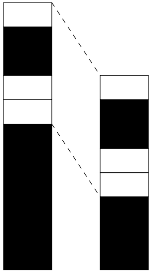
State equivalence

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

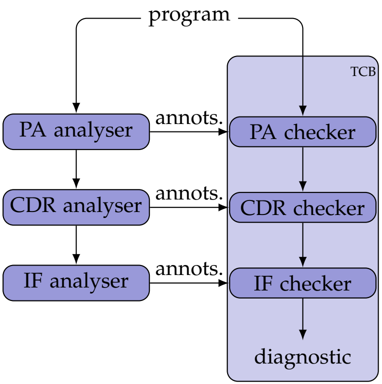
The type system:

- 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:

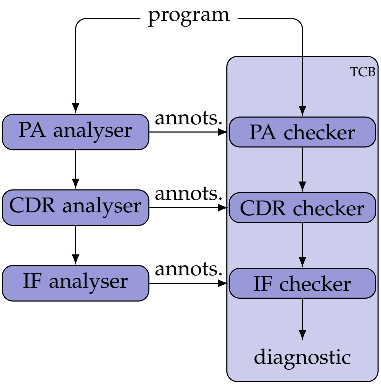
- objects
  (heap equivalence, allocator)
- exceptions
  (loss of precision)
- methods (extended signatures)



Adding objects, exceptions and methods

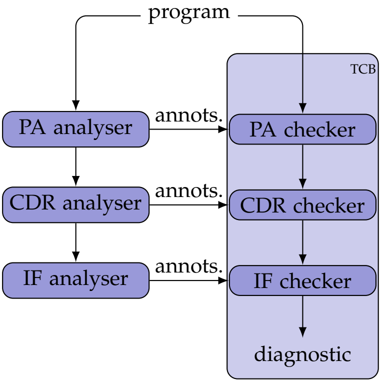
Three successive phases:

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*



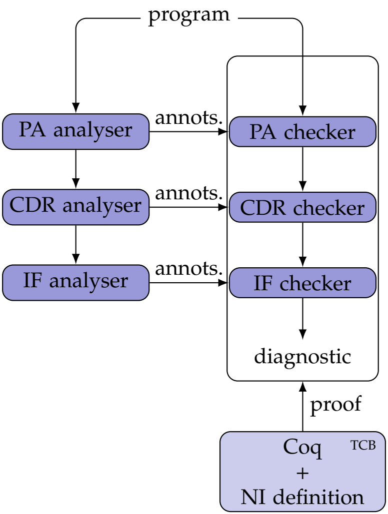
Adding objects, exceptions and methods

- 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

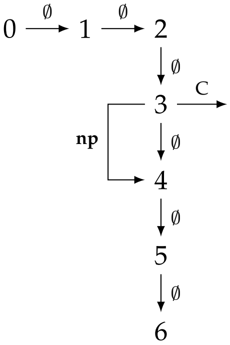
$$m : (x : L,\ y : H) \xrightarrow{H} \{\mathsf{n} : H,\ C : L,\ \mathbf{np} : H\}$$

- $k_h = H$: 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

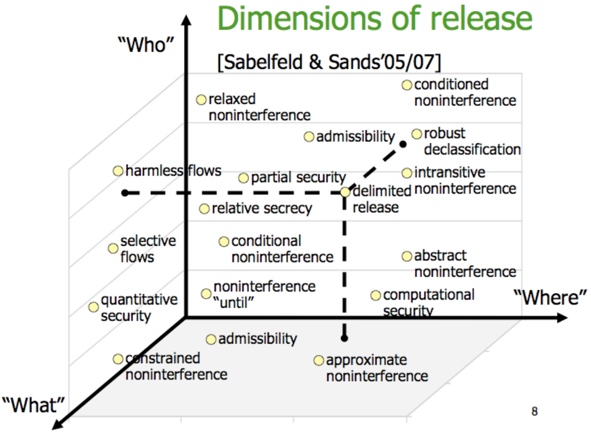
Towards realistic applications

Many features of missing to program realistic applications:

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

Declassification

- 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.

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

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

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

Local policies vs. declassify statements

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

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

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:

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

becomes

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

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

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

- 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

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

where

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

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

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:

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

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

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

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

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

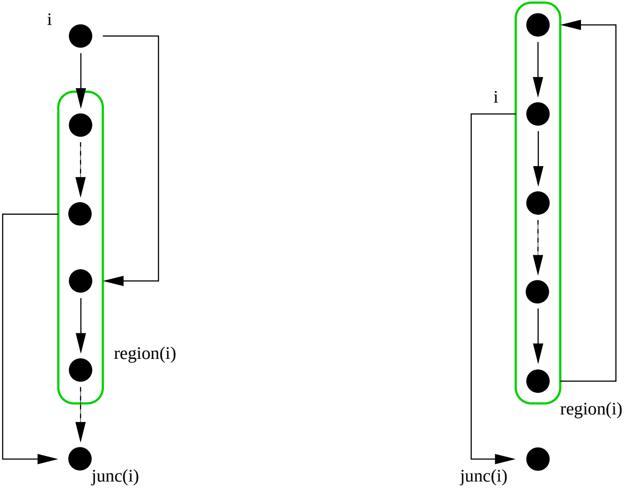
plus subtyping rules

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

Compiling statements

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

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:

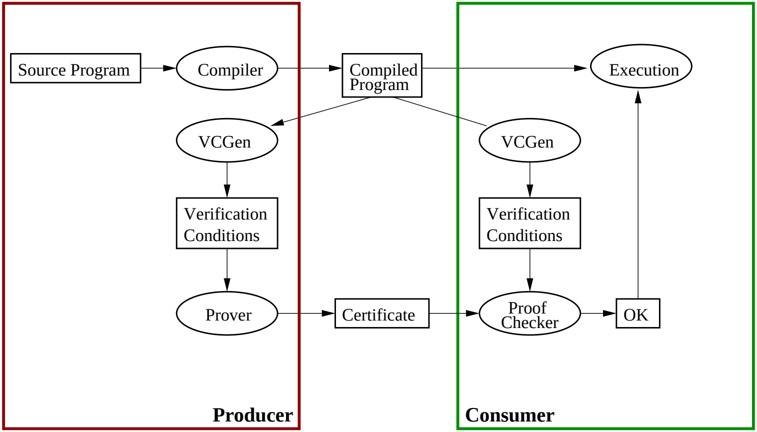
- regions satisfy SOAP and can be checked by region checker
- $[\![P]\!]$ can be verified by lightweight checker

The result also applies to

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

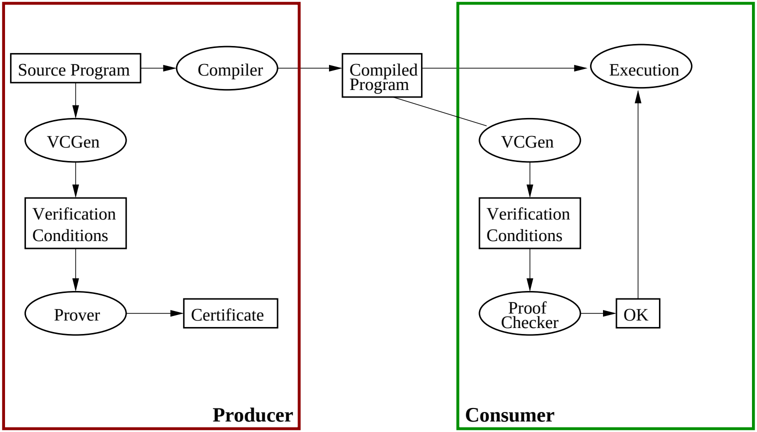
Motivation: source code verification

Traditional PCC

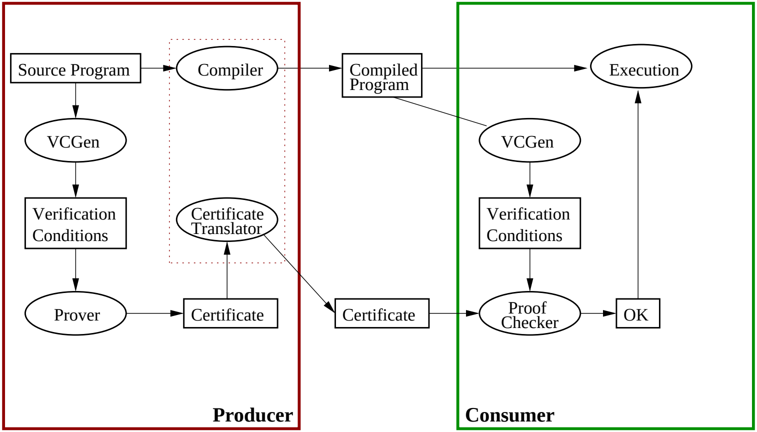


Motivation: source code verification

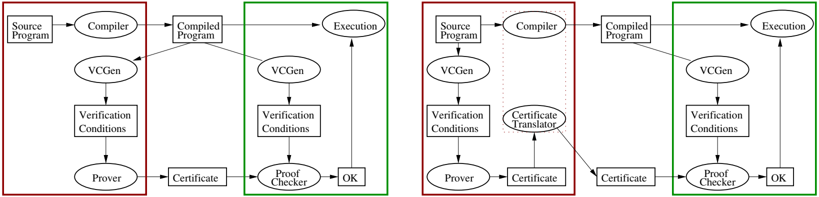
Source Code Verification



Certificate Translation



Certificate translation vs certifying compilation



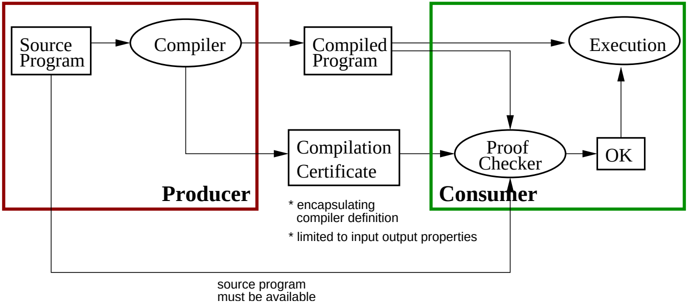
Conventional PCC		Certificate Translation
Automatically inferred invariants	**Specification**	Interactive
Automatic certifying compiler	**Verification**	Interactive source verification
Safety	**Properties**	Complex functional properties

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

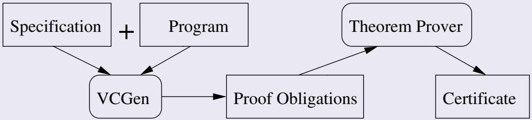
$\overline{\mathsf{ins}} ::= \mathsf{ins} \mid \langle \varphi, \mathsf{ins} \rangle$

- 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

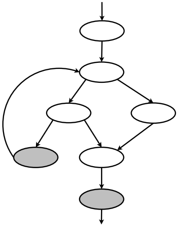
All infinite paths must go through an annotated program point

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

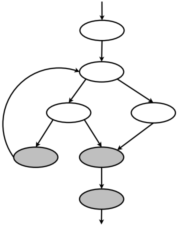


Weakest precondition calculus

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

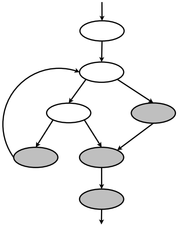
$$\begin{array}{lcll} \mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\ \mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise} \end{array}$$

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

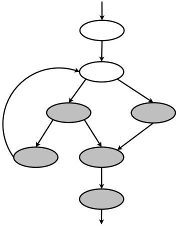
$$
\begin{array}{lcll}
\mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise}
\end{array}
$$

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

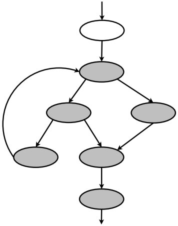
$$\begin{array}{lcll} \mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\ \mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise} \end{array}$$

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

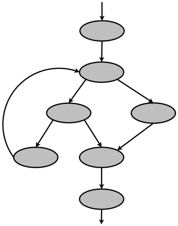
$$
\begin{array}{lcll}
\mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise}
\end{array}
$$

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

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

Weakest precondition

- if $P[k] = \mathsf{push}\ n$ then

$$\mathsf{wp}_i(k) = \mathsf{wp}_{\mathcal{L}}(k+1)[n/os[\top], \top/\top-1]$$

- if $P[k] = \mathsf{binop}\ op$ then

$$\mathsf{wp}_i(k) = \mathsf{wp}_{\mathcal{L}}(k+1)[os(\top-1)\ op\ os[\top]/os[\top], \top-1/\top]$$

- if $P[k] = \mathsf{load}\ x$ then

$$\mathsf{wp}_i(k) = \mathsf{wp}_{\mathcal{L}}(k+1)[x/os[\top], \top/\top-1]$$

- if $P[k] = \mathsf{store}\ x$ then

$$\mathsf{wp}_i(k) = \mathsf{wp}_{\mathcal{L}}(k+1)[os[\top]/x, \top-1/\top]$$

- if $P[k] = \mathsf{if}\ cmp\ l$ then

$$\begin{aligned}\mathsf{wp}_i(k) = & (os[\top-1]\ cmp\ os[\top] \Rightarrow \mathsf{wp}_{\mathcal{L}}(k+1)[\top-2/\top]) \\ & \wedge(\neg(os[\top-1]\ cmp\ os[\top]) \Rightarrow \mathsf{wp}_{\mathcal{L}}(l)[\top-2/\top])\end{aligned}$$

- if $P[k] = \mathsf{goto}\ l$ then $\mathsf{wp}_i(k) = \mathsf{wp}_{\mathcal{L}}(l)$
- if $P[k] = \mathsf{return}$ then $\mathsf{wp}_i(k) = \Psi[os[\top]/\mathsf{res}]$

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

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

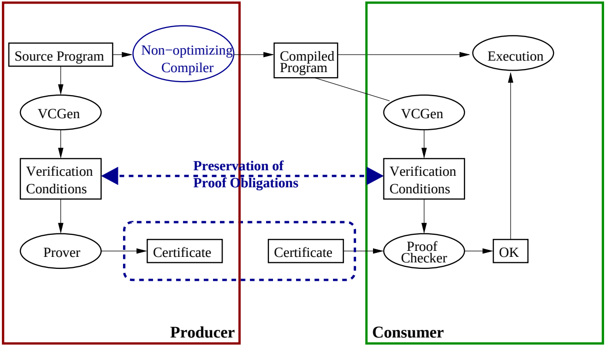
$$\frac{\mathsf{wp}_{\mathcal{S}}(i_t, \mathrm{post}) = \phi_t, \theta_t \quad \mathsf{wp}_{\mathcal{S}}(i_f, \mathrm{post}) = \phi_f, \theta_f}{\mathsf{wp}_{\mathcal{S}}(\mathsf{if}(t)\{i_t\}\{i_f\}, \mathrm{post}) = (t \Rightarrow \phi_t) \wedge (\neg t \Rightarrow \phi_t), \theta_t \cup \theta_f}$$

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

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

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)

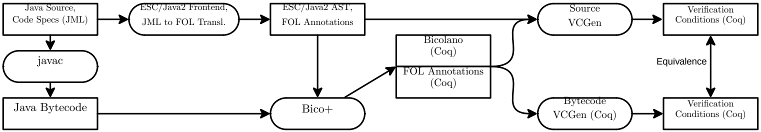
Reflective Proof Carrying Code

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

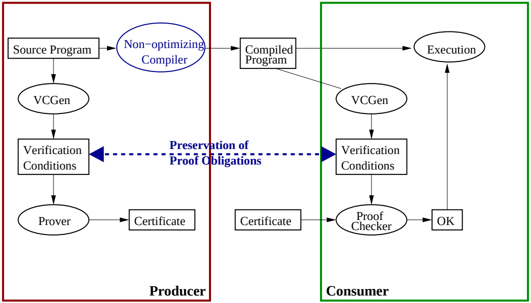
```
Lemma vc_equiv: vc_source <-> vc_bytecode.
```



The main tactic

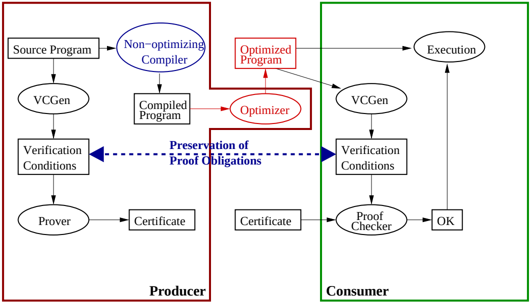
```
Ltac magickal :=
   repeat match goal with
   | [ |- forall lv: LocalVar.t, _ ] =>let lv := fresh "lv" in
                                         intro lv; mklvget lv 0%N
   | [ H: forall lv: LocalVar.t, _ |- _ ] => mklvupd MDom.LocalVar.empty 0%N
   | [ |- forall os: OperandStack.t, _ ] => intro
   | [ 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



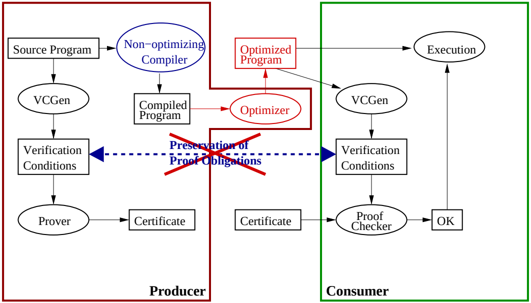
Proofs obligations might not be preserved

Optimizing Compilers



Proofs obligations might not be preserved

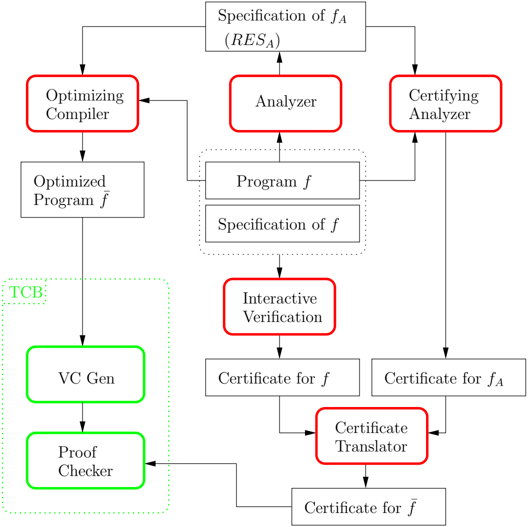
Optimizing Compilers



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

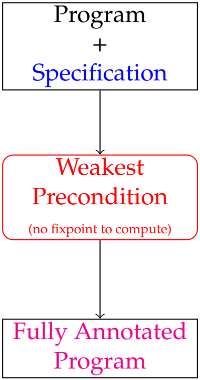
$\{j = 0\}$

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

$\quad i := c + i$

$\quad j := x * i;$

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



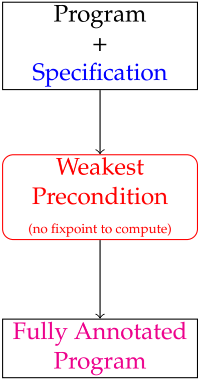
Motivating example

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

$\quad i := c + i$

$\quad j := x * i;$

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

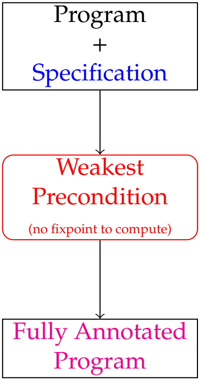


Motivating example

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

$\quad i := c + i$

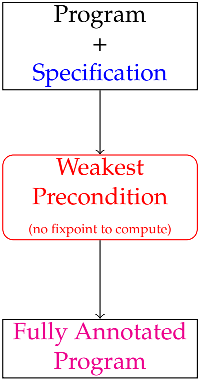
$\quad j := x * i;$
$\{j = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$
$endwhile;$
$\{n * b \leqslant j\}$



Motivating example

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

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



Motivating example

$\{j = 0\}$

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

$i := 0;$

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

$x := b + i;$

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

$while(i! = n)$

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

$\quad i := c + i$

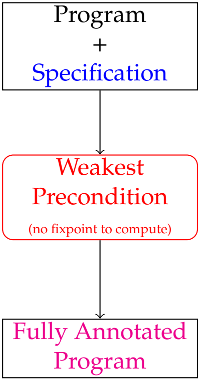
$\{x * i = x * i \wedge b \leqslant x \wedge 0 \leqslant i\}$

$\quad j := x * i;$

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

$endwhile;$

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



Motivating example

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

Set of Proof Obligations:

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

Constant propagation analysis

$\{j = 0\}$

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

$i := 0;$

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

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

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

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

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

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

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

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

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

$endwhile;$

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

Program transformation

$\{j = 0\}$

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

$i := 0;$

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

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

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

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

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

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

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

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

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

$endwhile;$

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

Program transformation

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

WP Computation of optimized program

$\{j = 0\}$

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

$i := 0;$

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

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

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

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

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

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

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

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

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

$endwhile;$

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

WP Computation of optimized program

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

WP Computation of optimized program

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

WP Computation of optimized program

$\{j = 0\}$

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

$i := 0;$

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

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

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

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

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

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

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

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

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

$endwhile;$

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

WP Computation of optimized program

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

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. $j = x * i \wedge b \leqslant x \wedge 0 \leqslant i \wedge i = n \Rightarrow n * b \leqslant j$

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

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

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

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

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

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

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

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

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

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

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

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

S_1		S_1
$\{\varphi_1\}$		$\{\varphi_1 \wedge \psi_1\}$
S_2		S_2
$\{\varphi_2\}$	$\leadsto$	$\{\varphi_2 \wedge \psi_2\}$
S_3		S_3
$\{\varphi_3\}$		$\{\varphi_3 \wedge \psi_3\}$
• $\varphi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2)$		• $\varphi_1 \wedge \psi_1 \Rightarrow \mathsf{wp}(S_1, \varphi_2) \wedge \mathsf{wp}(S_1, \psi_2)$
• $\varphi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3)$		• $\varphi_2 \wedge \psi_2 \Rightarrow \mathsf{wp}(S_2, \varphi_3) \wedge \mathsf{wp}(S_2, \psi_3)$

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

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

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

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

Translation consists of:

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

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

Translation consists of:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

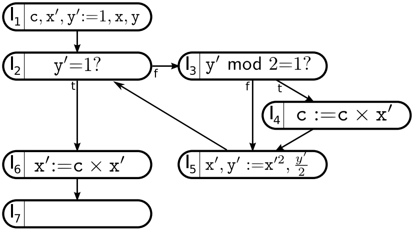
Benefits of generalization

- 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

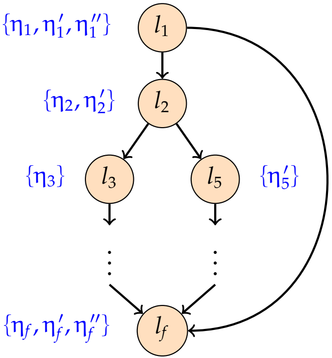
$$c := 1$$
$$x' := x$$
$$y' := y$$
$$\textit{while } (y' \neq 1) \textit{ do}$$
$$\quad \textit{if } (y' \textit{ mod } 2 = 1) \textit{ then}$$
$$\quad\quad c := c \times x'$$
$$\quad \textit{fi}$$
$$\textit{done}$$
$$x' = x' \times c$$



Program: directed graph

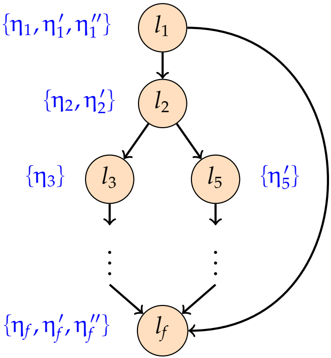
- Nodes denoting execution points ($\mathcal{N}$).
- Edges denoting possible transitions between nodes ($\mathcal{E}$).

Program semantics

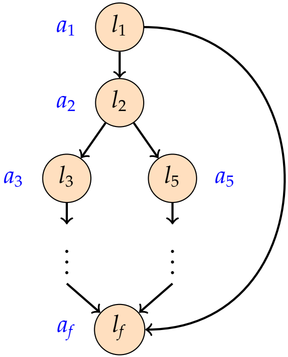


Abstract Interpretation

Program semantics

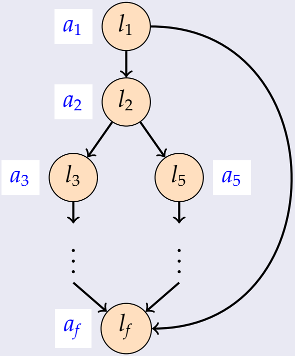


Abstract representation



Solution of a Forward Abstract Interpretation

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

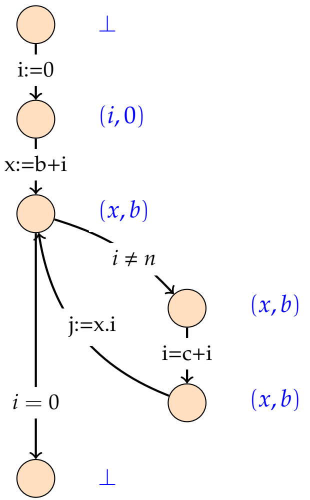


$\{a_1, a_2, \ldots, a_f\}$ a solution of $(\mathbf{D}, T)$ if:

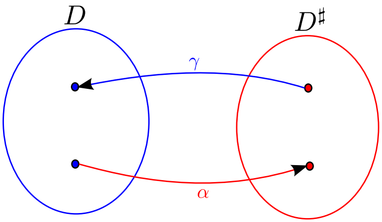
$$T_{\langle l_1, l_2 \rangle}(a_1) \sqsubseteq a_2$$
$$T_{\langle l_2, l_5 \rangle}(a_2) \sqsubseteq a_5$$
$$T_{\langle l_1, l_f \rangle}(a_1) \sqsubseteq a_f$$
$$\ldots$$

Example of decidable solution

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



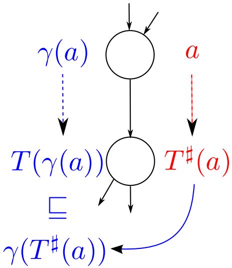
Galois connections capture notion of imprecision



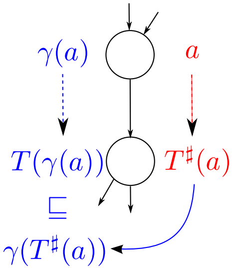
In the following (intuition):

- (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



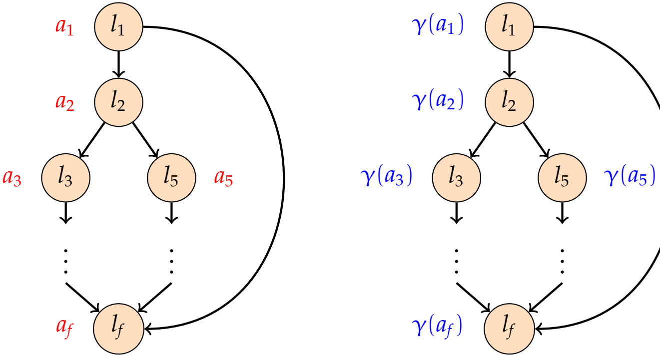
$$T(\gamma(a)) \sqsubseteq \gamma(T^\sharp(a))$$



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

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 $\langle i,j \rangle$

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

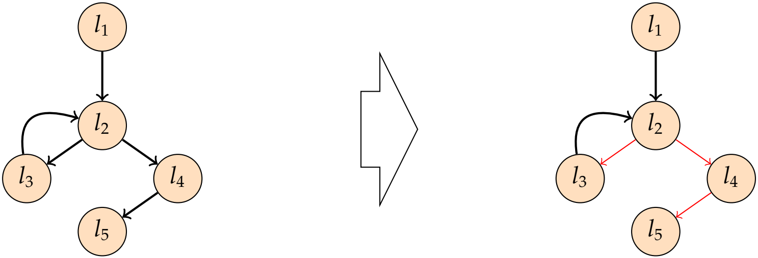
if $(\{a_1 \ldots a_n\}, c_a)$ and $(\{b_1 \ldots b_n\}, c_b)$ are certified solutions of D, then $(\{a_1 \sqcap b_1 \ldots a_n \sqcap b_n\}, c_a \oplus c_b)$ is a certified solution.

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

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

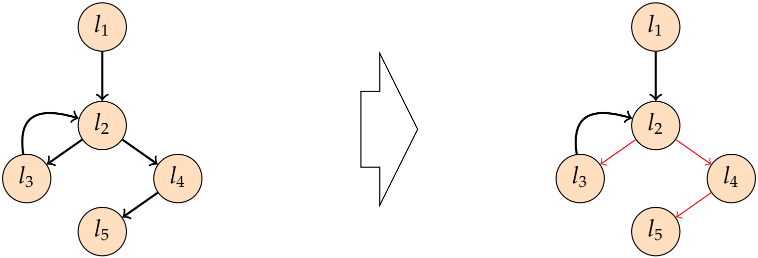
then $(\{\gamma(a_1) \ldots \gamma(a_n)\}, c)$ is a certified solution of (D, T) [for some c].

Program Transformation



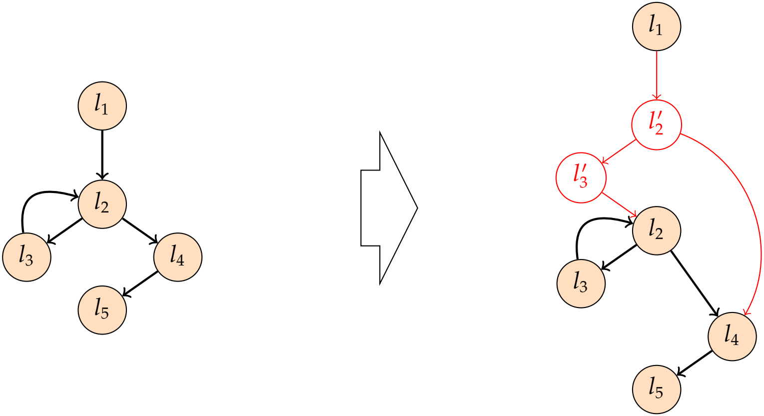
- $T_e \mapsto T'_e$, $e \in \mathcal{E}$
- a proof of $T'_{\langle l_2, l_3 \rangle}(\_) \sqsubseteq a_3 \sqcap T_{\langle l_2, l_3 \rangle}(\_)$
- const and copy propag / loop induction var strength reduction / common. subexpr elimination / etc.

Program Transformation



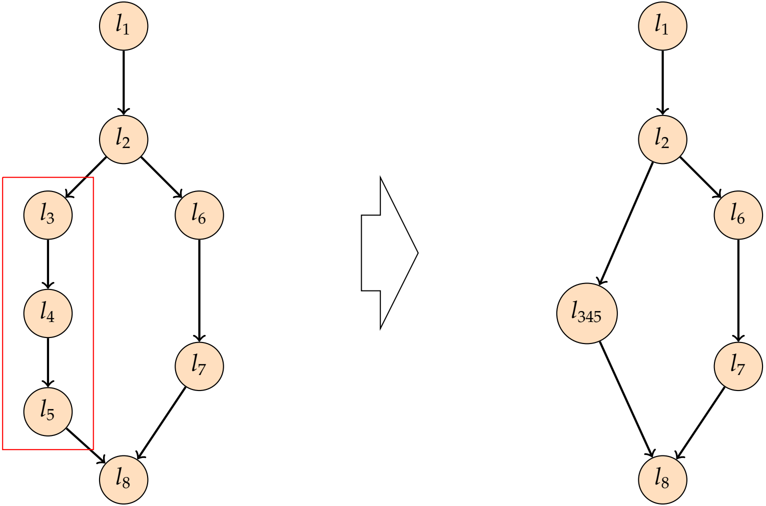
- $T_e \mapsto T'_e, e \in \mathcal{E}$
- a proof of $T'_{\langle l_2, l_3 \rangle}(\_) \sqsubseteq a_3 \sqcap T_{\langle l_2, l_3 \rangle}(\_)$
- const and copy propag / loop induction var strength reduction / common. subexpr elimination / etc.

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

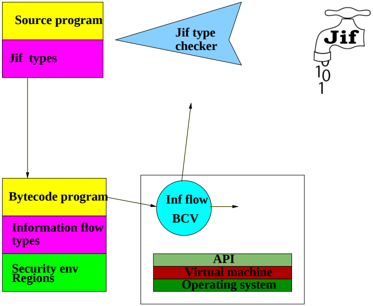
Two verification methods for bytecode and their relation to verification methods for source code

- 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

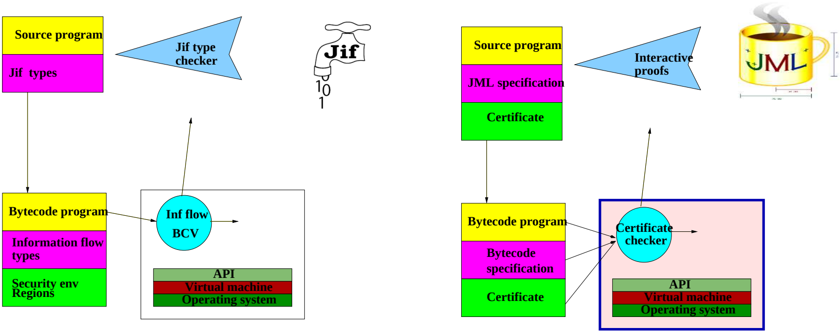
- Type system for information flow based confidentiality policies
- Verification condition generator for logical specifications



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



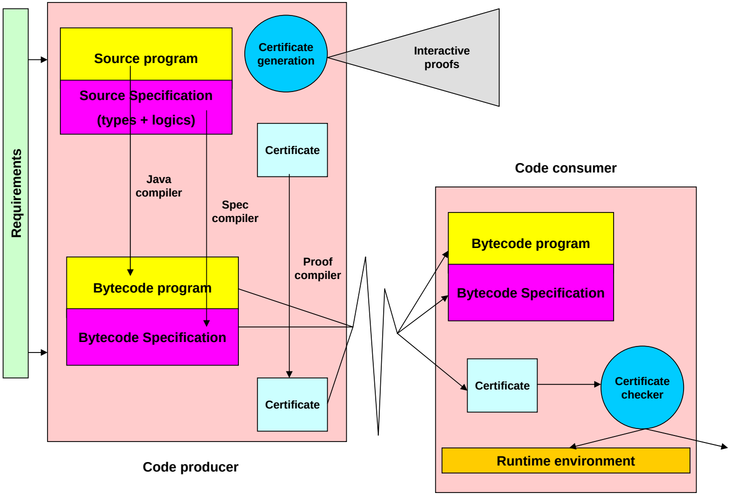
Deployment of secure mobile code can benefit from:

- 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