Compilation of certificates

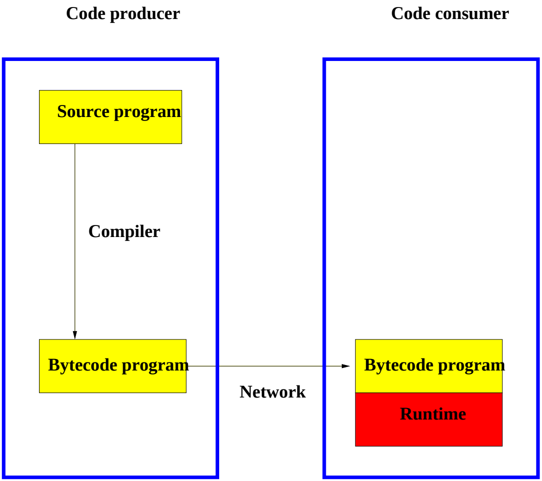
Gilles Barthe

INRIA Sophia-Antipolis Méditerranée, France

Motivation

- Mobile code is ubiquitous
  - Large distributed networks of JVM devices
    - aimed at providing a global and uniform access to services
- Security is a central concern:
  - applications manipulate sensitive data stored on devices
  - communications must be secured
- Issues:
  - uniform access to services vs. heterogeneity of devices
  - flexibility of computational infrastructure vs. rigid security architecture
  - platform must be correct (VM, API,. . .)
  - lack of appropriate security mechanisms to guarantee security on consumer side

Security challenge



Proof Carrying Code

Downloaded components come equipped with certificates, where certificates:

- are condensed and formalized mathematical proofs/hints
- are self-evident and unforgeable
- can be checked efficiently

Proof Carrying Code

Downloaded components come equipped with certificates, where certificates:

- are condensed and formalized mathematical proofs/hints
- are self-evident and unforgeable
- can be checked efficiently

pgflastimage

Proof Carrying Code

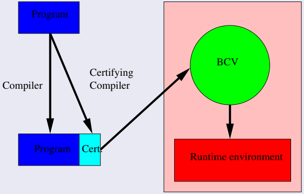
Downloaded components come equipped with certificates, where certificates:

- are condensed and formalized mathematical proofs/hints
- are self-evident and unforgeable
- can be checked efficiently

pgflastimage

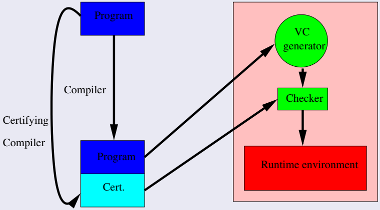
Flavors of Proof Carrying Code

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

Contents

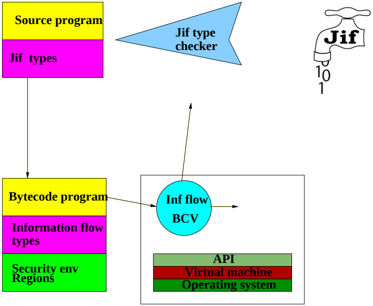
The objective of the course is to present verification methods for bytecode and relate them to verification methods for source code

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

Contents

The objective of the course is to present verification methods for bytecode and relate them to verification methods for source code

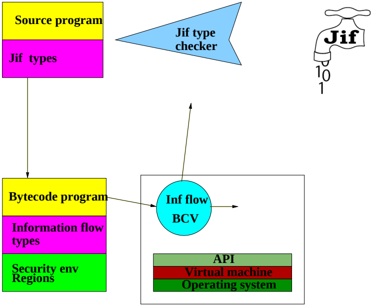
- Type system for information flow based confidentiality policies
- Verification condition generator for logical specifications

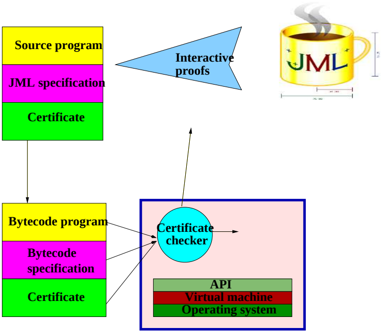


Contents

The objective of the course is to present verification methods for bytecode and relate them to verification methods for source code

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





Mobius: Mobility, Ubiquity, Security

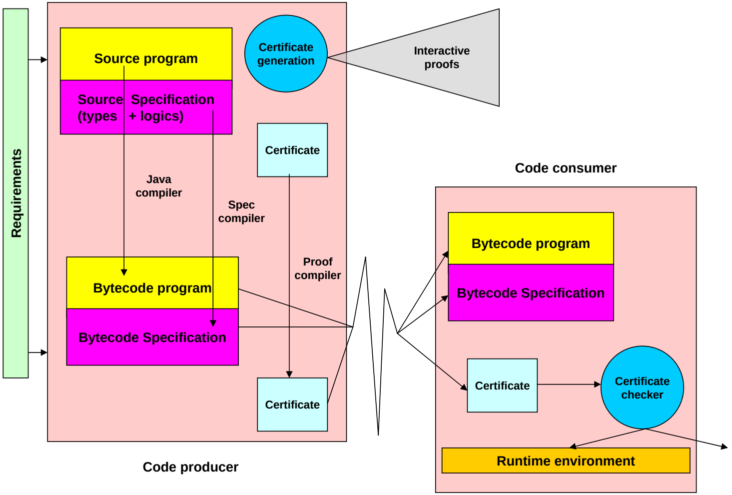
Main goals:

- develop basic technologies (type systems and logics) for static enforcement of expressive policies at application level:
  - confidentiality, integrity, resource usage
  - logical specifications
- build a Proof Carrying Code infrastructure that integrates these basic technologies
- use proof assistants to achieve the highest guarantees for security mechanisms

Mobius

INRIA
ETH Zürich
LMU Münich
RU Nijmegen
U. Edinburgh
Chalmers U.
Tallinn U.
Imperial College
UC Dublin
U. Warsaw
UP Madrid
TLS
SAP Research
France Telecom
Trusted Logic
TU Darmstadt

The Mobius view

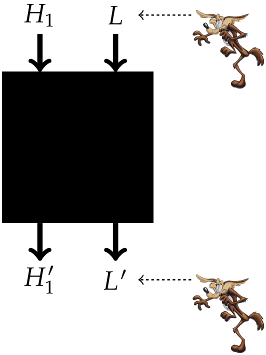


Part 1: Information flow typing

- G. Barthe, D. Naumann and T. Rezk, *Deriving an Information Flow Checker and Certifying Compiler for Java*, Security and Privacy 2006
- G. Barthe, D. Pichardie and T. Rezk, *A Certified Lightweight Non-Interference Java Bytecode Verifier*, ESOP'07
- G. Barthe, T. Rezk, A. Russo and A. Sabelfeld, *Security of Multi-Threaded Programs by Compilation*, ESORICS'07

Non-interference

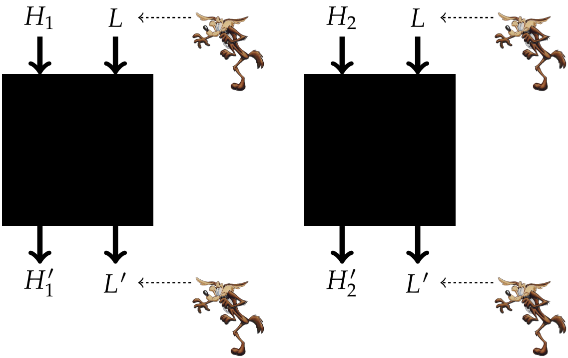
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidential   Low = public

Non-interference

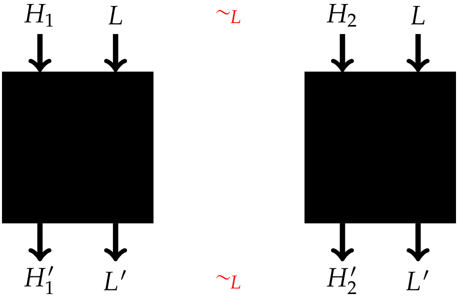
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



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

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

Programs

A program is an array of instructions:

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

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

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

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

Semantics: rules

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

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

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

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

$$\frac{P[i] = \mathsf{ifeq}\ j}{\langle i, \rho, 0 :: s\rangle \rightsquigarrow \langle j, \rho, s\rangle}$$

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad n \neq 0}{\langle i, \rho, n :: s\rangle \rightsquigarrow \langle i+1, \rho, s\rangle}$$

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

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

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

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

- $S \vdash P$ 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)

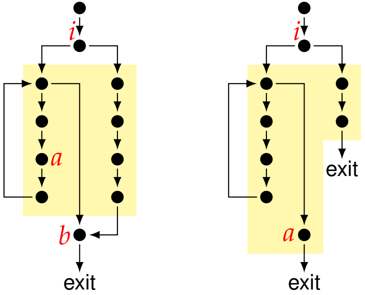
A program point j is in a *control dependence region* of a branching point i if

- 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 $j \in region(i)$

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

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

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

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

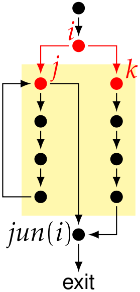
SOAP3: for all program points i, j, if $j \in region(i)$ and $j \mapsto$ then $jun(i)$ is undefined.

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

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

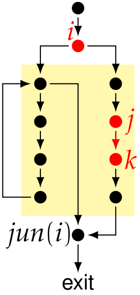
SOAP2: for all program points i, j, k, if $j \in region(i)$ and $j \mapsto k$, then either $k \in region(i)$ or $k = jun(i)$;

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



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

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

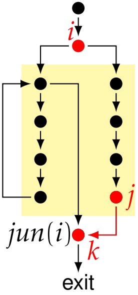


CDR soundness is ensured by local conditions (instead of path properties) using *region* $\in \mathcal{P} \rightarrow \wp(\mathcal{P})$ and *jun* $\in \mathcal{P} \rightharpoonup \mathcal{P}$.

SOAP1: for all program points i and all successors j, k of i ($i \mapsto j$ and $i \mapsto k$) such that $j \neq k$ (i is hence a branching point), $k \in \mathit{region}(i)$ 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.

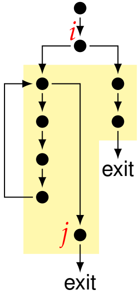


CDR soundness is ensured by local conditions (instead of path properties) using $region \in \mathcal{P} \to \wp(\mathcal{P})$ and $jun \in \mathcal{P} \rightharpoonup \mathcal{P}$.

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.



Transfer rules

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

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

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

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

Soundness

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

Proof:

Soundness

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

Proof:

Lightweight checking algorithm

- Code provided with:
  - regions (verified by a region checker),
  - security environment
  - type annotations for junction points (most often empty)
- Program entry point is typed with the empty stack
- Propagation
  - Pick a program point i annotated with st
  - Compute st' such that $i \vdash st \Rightarrow st'$. If there is no st', then reject program.
  - If st' exists, then for all successors j of i
    - if j is not yet annotated, annotated it with st'
    - if j is annotated with st'', check that $st' \leqslant st''$. If not, reject program

Type-preserving compilation

- Source information flow type system offers a tool for developing secure applications, but does not directly address mobile code
- Bytecode verifier provides information flow assurance to users
- Reconcile both views by showing that typable programs are compiled into typable programs

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

Must also compile regions and security environment

Source language

- Programs are commands

$$c ::= x := e \mid \mathsf{if}(e)\{c\}\{c\} \mid \mathsf{while}(e)\{c\} \mid c; c \mid \mathsf{skip} \mid \mathsf{return\ e}$$

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

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

$$\frac{e : pc \quad [pc] \vdash c_1 \quad [pc] \vdash c_2}{[pc] \vdash \mathsf{if}(e)\{c_1\}\{c_2\}} \qquad \frac{e : pc \quad [pc] \vdash c}{[pc] \vdash \mathsf{while}(e)\{c\}} \qquad \frac{e : k \quad k \sqcup pc \leqslant k_{\mathrm{ret}}}{[pc] \vdash \mathsf{return}\ e}$$

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

Compiler

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

Semantics-preserving compilation

Compiler correctness

$P, \mu \Downarrow \nu, v$ implies $[\![P]\!], \mu \Downarrow \nu, v$

Consequences:

- Source programs non-interfering iff their compilation is non-interfering

$$\forall P, P \text{ is non-interfering} \iff [\![P]\!] \text{ is non-interfering}$$

- Type-preservation entails soundness of source type system:

$$\forall P, \vdash P \implies P \text{ is non-interfering}$$

However, preservation of typing is not a consequence of compiler correctness

Compiling regions and security environment

- Regions are compiled by induction on structure of programs
- Security environment and type annotations computed from typing derivation

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

load y_H	L		ϵ
ifeq 6	L		$H : \epsilon$
push 1	H	$\in region(2)$	ϵ
store x	H	$\in region(2)$	$H : \epsilon$
goto 8	H	$\in region(2)$	ϵ
push 2	H	$\in region(2)$	ϵ
store x	H	$\in region(2)$	$H : \epsilon$
push 3	L	$= jun(2)$	ϵ
store x'	L		$L : \epsilon$
push 2	L		ϵ
return	L		$L : \epsilon$

Preservation of information flow types

- For our (non-optimizing) compiler:
  If P is typable, then $[\![P]\!]$ is typable wrt security environment, regions, and type annotations generated by extended compiler. Furthemore, generated regions satisfy SOAP.
- For optimizing compilers, type preserving compilation may fail:

  $$x_H := n_1 * n_2; y_L := n_1 * n_2 \quad \Longrightarrow x_H := n_1 * n_2; y_L := x_H$$

  There may be easy fixes

Concurrency

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

$$\mathsf{if}(y_H > 0)\{\mathsf{skip};\mathsf{skip}\}\{\mathsf{skip}\}; x_L := 1 || \mathsf{skip};\mathsf{skip};\ \mathsf{x_L} := 2$$

  Using round robin scheduler with time slice one:
  - if $y_H > 0$ then $x_L := 1$
  - if not $y_H > 0$ then $x_L := 2$
- 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 hidden (currently executing under the scope of a high branching instruction), 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)

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

Multi-threaded language

- Instruction for dynamic thread creation $\mathsf{start}\ i$
- States $\langle \rho, \lambda \rangle$ where λ associates to each active thread a pair $\langle i, s \rangle$.
- Semantics lifted from sequential fragment

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

where

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

Policy and type system

- Policy and type system similar to sequential fragment
- Transfer rules inherited from sequential fragment

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

- Assume the scheduler is secure, type soundness and type preservation can be lifted from sequential language:
  - Type soundness: same proof techniques (using extended SOAP properties)
  - Type preservation: parallel composition typed in the naive way

$$\frac{[pc] \vdash P \qquad [pc] \vdash Q}{[pc] \vdash P||Q}$$

  compiler generates security environment that prevents internal timing leaks

Adding objects, exceptions and methods

We have formally proved in Coq the soundness of information flow type system for a sequential JVM-like language, and extracted an information flow checker.

Main issue is with exceptions:

- loss of precision due to explosion of control flow
- regions are parametrized by exceptions
- more complex signatures and typing rules
- for type-preserving compilation, loss of structure at source level

Adding objects, exceptions and methods

We have formally proved in Coq the soundness of information flow type system for a sequential JVM-like language, and extracted an information flow checker.

Main issue is with exceptions:

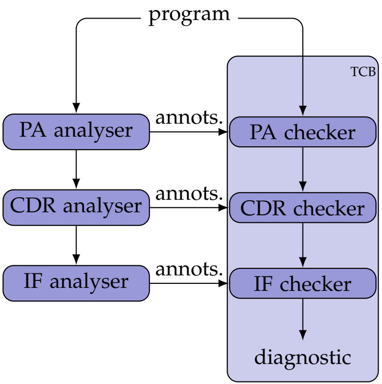
- loss of precision due to explosion of control flow
- regions are parametrized by exceptions
- more complex signatures and typing rules
- for type-preserving compilation, loss of structure at source level

There are also interesting issues wrt dynamic object creation:

- heap L-equivalence
- allocator may leak information

A three-stage typing system

1. preliminary analysis in order to reduce control flow graph
   - null pointers
   - array accesses
   - ...
2. CDR analyser computes *control dependence regions*
3. IF (Information Flow) analyser computes a *security environment* and a *type*



Information flow type system

Type annotations required on programs:

- $ft : \mathcal{F} \to \mathcal{S}$ attaches security levels to fields,
- 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
- 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

Typing judgment

General form

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

Invokation

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \vec{k_a'} \xrightarrow{k_h'} \vec{k_r'} \\ k \sqcup k_h \sqcup se(i) \leqslant k_h' \qquad k \leqslant \vec{k_a'}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leqslant \vec{k_a'}[i+1] \\ e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \cup \{\mathbf{np}\} \qquad \forall j \in region(i,e),\ k \sqcup \vec{k_r'}[e] \leqslant se(j) \qquad \mathsf{Handler}(i,e) = t \end{array}}{\Gamma, region, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow (k \sqcup \vec{k_r'}[e]) :: \varepsilon}$$

other 60 typing rules...

Example of typable program

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

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

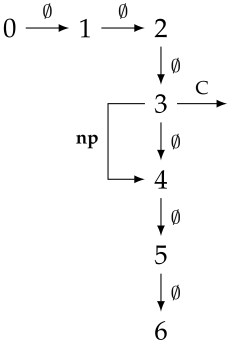
$$m : (x : L,\ y : H) \xrightarrow{H} \{\mathsf{n} : H,\ C : L,\ \mathbf{np} : H\}$$

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

Justifying fine grain treatment of exceptions

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

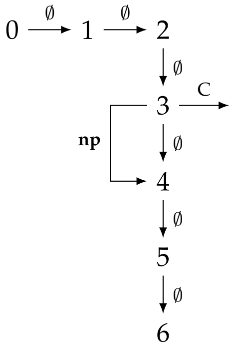
0 : load o_L
1 : load y_H
2 : load x_L
3 : invokevirtual m
4 : store z_H
5 : push 1
6 : store t_L
handler : $[0,3]$, NullPointer $\rightarrow 4$



Justifying fine grain treatment of exceptions

```
try {z = o.m(x,y);} catch (NullPointerException 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 $\rightarrow 4$



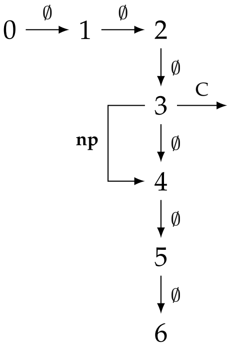
Naive treatment of exceptions

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

Justifying fine grain treatment of exceptions

```
try {z = o.m(x,y);} catch (NullPointerException 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 $\rightarrow 4$

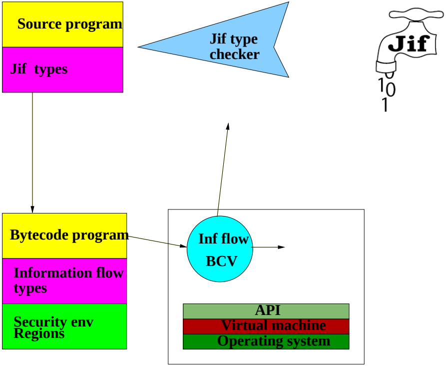


Treating each exception separately

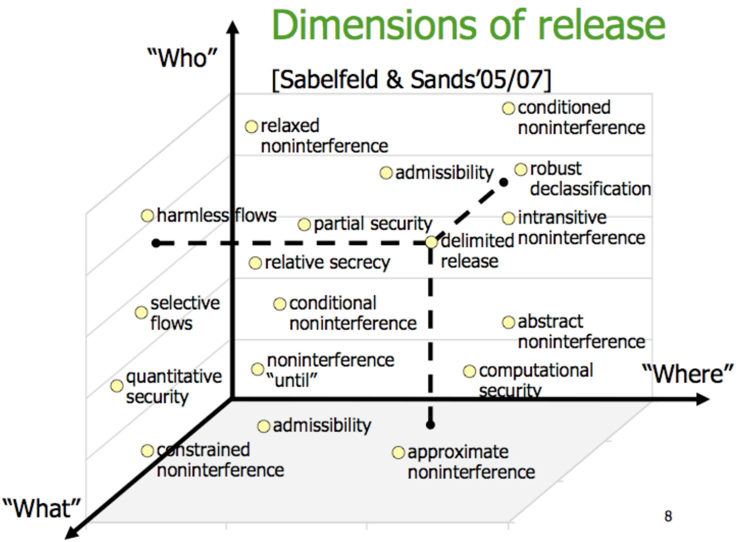
- [4,5,6] is a low region: $t_L = 1$ is accepted

Summary

- We have developed:
  - Sound information flow bytecode verifier for sequential fragment of JVM
  - Type-preserving compiler for Java
- Next goal is to provide support for realistic applications:
  - more flexible type system
  - more flexible policies
    - Trusted declassifier
    - Cryptography
- Other goals
  - certifying compilation
  - distribution by compilation



Declassification (from A. Sabelfeld)



Final remarks on machine-checked proofs

- Implementing an information flow type checker for JVM is a non-trivial task
- Do you trust your implementation? Do you trust the non-interference proof?

We have used the Coq proof assistant

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

Machine-checked proof: structure

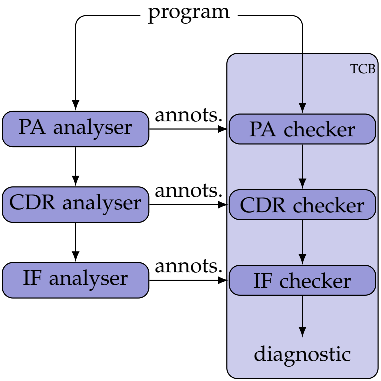
1. Basis : JVM program and small-step semantics formalisation (Bicolano)
2. Intermediate semantics:
   - operates on annotated programs
   - method calls are big-step (simpler definition of $\sim_L$ without callstacks; inappropriate for multi-threading)
3. Implementation and correctness proof of the CDR checker
4. Implementation and correctness proof of the information flow type system

Human effort

- about 20,000 lines of definitions and proofs with a reasonable Coq style programming,
- about 3,000 lines are only there to define the JVM semantics

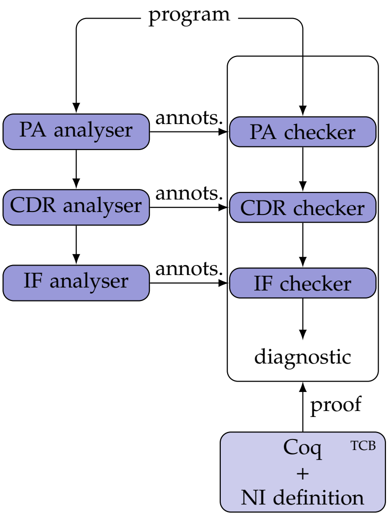
Architecture revisited

Since we prove these checkers in Coq, TCB is in fact relegated to Coq and the formal definition of non-interference.



Architecture revisited

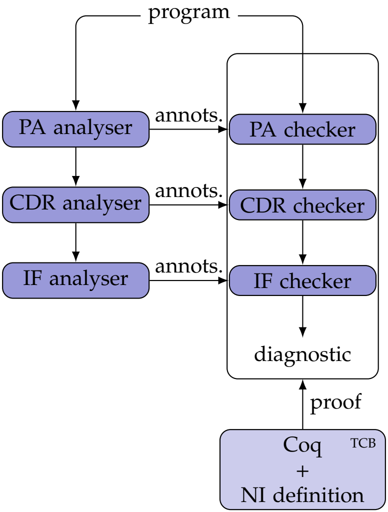
Since we prove these checkers in Coq, TCB is in fact relegated to Coq and the formal definition of non-interference.



Architecture revisited

Since we prove these checkers in Coq, TCB is in fact relegated to Coq and the formal definition of non-interference.

- Similar to Appel's Foundational PCC
- We exploit reflection to achieve small certificates



Compilation of certificates

Gilles Barthe

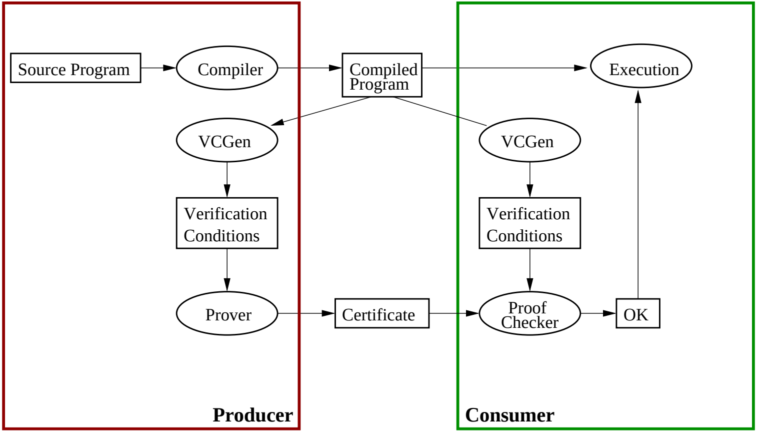
INRIA Sophia-Antipolis Méditerranée, France

Part 2: Verification condition generation

- G. Barthe, T. Rezk and A. Saabas, *Preservation of proof obligations*, FAST'05
- G. Barthe, B. Grégoire and M. Pavlova, *Preservation of proof obligations for Java*, 2007
- G. Barthe, B. Grégoire, C. Kunz and T. Rezk, *Certificate translation for optimizing compilers*, SAS'06

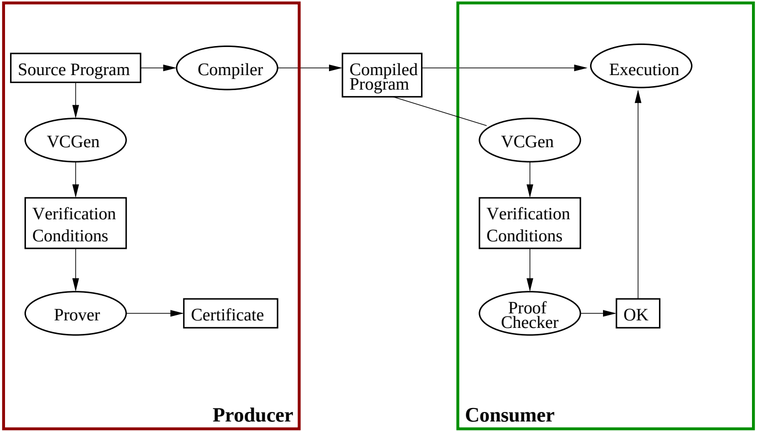
Motivation: source code verification

Traditional PCC



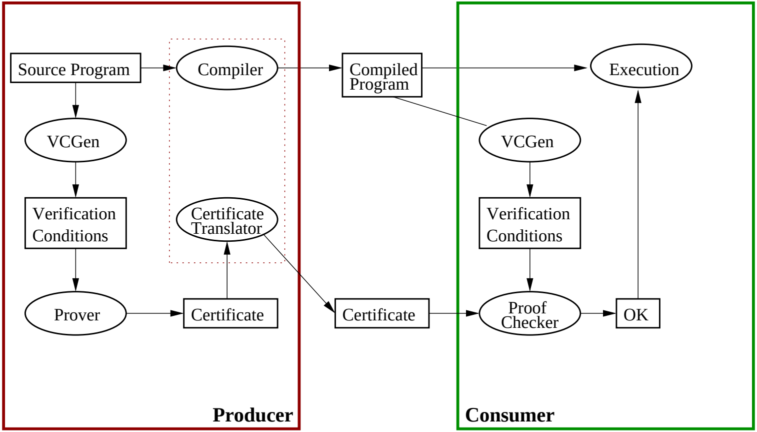
Motivation: source code verification

Source Code Verification

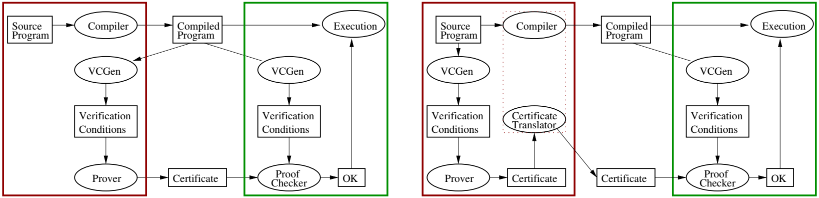


Motivation: source code verification

Certificate Translation



Certificate translation vs certifying compilation



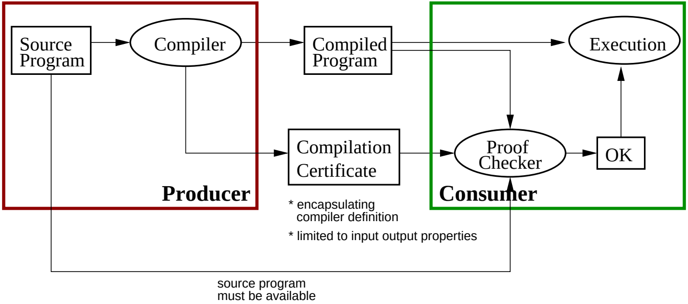
Conventional PCC		**Certificate Translation**
Automatically inferred invariants	**Specification**	Interactive
Automatic certifying compiler	**Verification**	Interactive source verification
Safety	**Properties**	Complex functional properties

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

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

- 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

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

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

Possibly annotated instructions

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

- 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

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

- 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

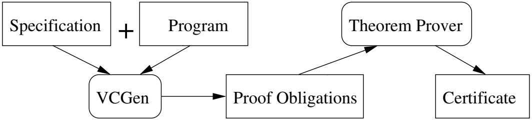
$\overline{\mathrm{ins}} ::= \mathrm{ins} \mid \langle \varphi, \mathrm{ins} \rangle$

- 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

VCGen

A verification Condition Generator (VCGen):

1. fully annotates a program
2. extracts a set of proof obligations



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

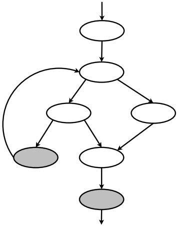
All infinite paths must go through an annotated program point

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

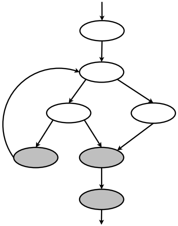


Weakest precondition calculus

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 $\mathrm{wp}_{\mathcal{L}}(k)$ of program point k

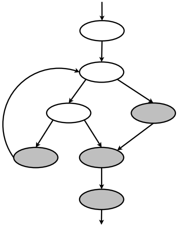
$$
\begin{array}{lcll}
\mathrm{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\
\mathrm{wp}_{\mathcal{L}}(k) & = & \mathrm{wp}_i(k) & \text{otherwise}
\end{array}
$$

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 $\mathrm{wp}_{\mathcal{L}}(k)$ of program point k

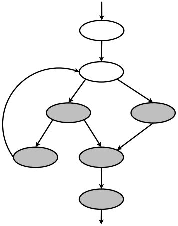
$$\begin{array}{lcll} \mathrm{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\ \mathrm{wp}_{\mathcal{L}}(k) & = & \mathrm{wp}_i(k) & \text{otherwise} \end{array}$$

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 $\mathrm{wp}_{\mathcal{L}}(k)$ of program point k

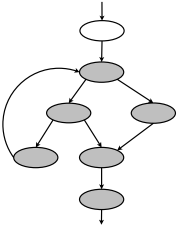
$$
\begin{array}{lcll}
\mathrm{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\
\mathrm{wp}_{\mathcal{L}}(k) & = & \mathrm{wp}_i(k) & \text{otherwise}
\end{array}
$$

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 $\mathrm{wp}_{\mathcal{L}}(k)$ of program point k

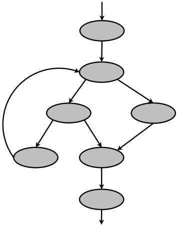
$$
\begin{array}{rcll}
\mathrm{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } P[k] = \langle \phi, i \rangle \\
\mathrm{wp}_{\mathcal{L}}(k) & = & \mathrm{wp}_i(k) & \text{otherwise}
\end{array}
$$

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 $\mathrm{wp}_{\mathcal{L}}(k)$ of program point k

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

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

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

Assertions

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

$$\begin{array}{lr} \{\mathit{true}\} & \\ \text{push } 5 & 5 = 5 \\ \text{store } x & \text{os}[\top] = 5 \\ \{x = 5\} & \end{array}$$

Assertions

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

Stack indices

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

Expressions

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

Assertions

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

$$\begin{array}{ll}\{true\} & \\ \text{push } 5 & 5 = 5 \\ \text{store } x & \mathrm{os}[\top] = 5 \\ \{x = 5\} & \end{array}$$

Weakest precondition

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

Verification conditions

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

- Precondition implies the weakest precondition of entry point:

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

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

$$\varphi \Rightarrow \mathrm{wp}_{i}(k)$$

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

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

Furthermore, all intermediate assertions are verified

Proof idea: if $s \leadsto s'$ and $s \cdot \mathrm{pc} = k$ and $s' \cdot \mathrm{pc} = k'$,

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

Source language

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

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

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

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

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

Preservation of proof obligations

Non-optimizing compiler

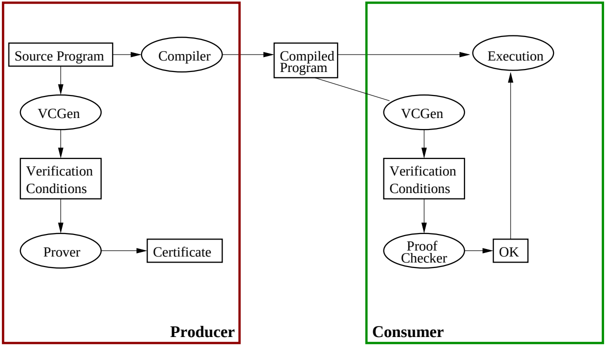
Syntactically equal proof obligations

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

Preservation of proof obligations
Non-optimizing compiler

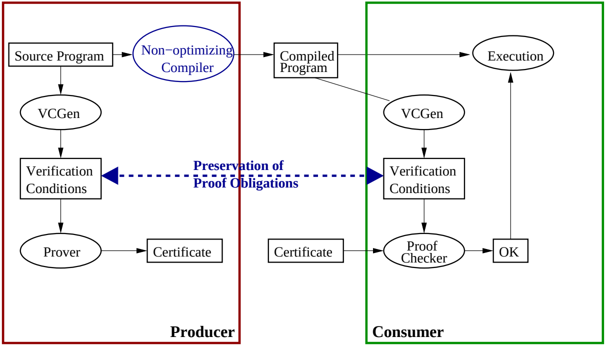
Syntactically equal proof obligations

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



Syntactically equal proof obligations

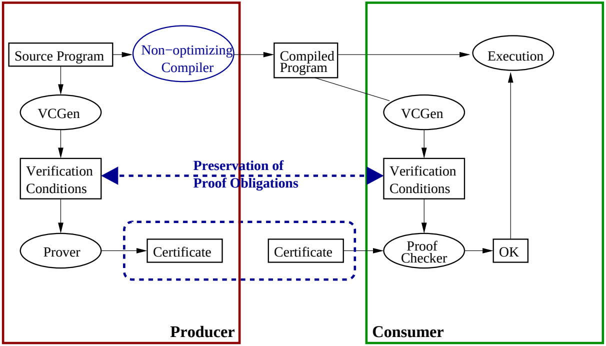
$$\mathrm{PO}(P, \phi, \psi) = \mathrm{PO}([\![P]\!], \phi, \psi)$$



Preservation of proof obligations
Non-optimizing compiler

Syntactically equal proof obligations

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



Adding objects, exceptions, methods

We have formally proved in Coq the soundness of VC generator for a sequential JVM-like language, and proved (almost) PPO for Java

Main points:

- Methods:
  - pre- and post-conditions to ensure modular reasoning
  - behavioral subtyping: method overriding should preserve spec
- Exceptions:
  - loss of precision due to explosion of control flow: preliminary analyses (as in information flow)
  - exceptional postconditions must be considered
- For PPO: renaming, booleans, simple optimizations

However:

- many challenges in verification of sequential OO programs
- concurrency (semantics and verification) is another challenge

Remark on machine-checked proof

- Implementing a verification condition generator for JVM is a non-trivial task
- Do you trust your implementation? And the soundness proof?

We have used the Coq proof assistant

- to formally specify the verification condition generator,
- to mechanically prove that valid proof obligations imply validity of annotations.

The development is structured in two layers:

1. Basis : JVM program and small-step semantics formalisation (Bicolano)
2. Intermediate semantics:
    - operates on annotated programs
    - method calls are big-step

Tools: Java Modeling Language

There is a range of tools that support deductive verification of Java applications. Many tools use JML as specification language.

- Annotation language for Java
- pre- and post-conditions and invariants written as special comments
- Uses Java-like notation
- Annotations are side-effect-free Java expressions + some extra keywords (`\exists`, `\forall`, `\old(−)`, `\result`...)
- Developed by Leavens et.al., Iowa State University
- Different tools available to validate, reason or generate JML annotations

Example

```
/*@ exceptional_behavior
  @   requires  arg == null;
  @   signals   (NullPointerException) true;
  @ also
  @ behavior
  @   requires  arg != null;
  @   ensures   \result == arg[0];
  @   signals   (IndexOutOfBoundsException)
  @                 arg.length == 0;
  @*/
Object firstElement (Object [] arg) {
  return arg[0];
}
```

Tools: verification

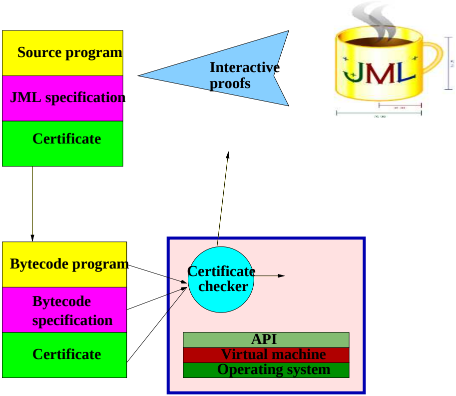
Many tools to validate Java applications annotated with JML: testing, run-time checking, model-checking. . .
and deductive verification tools:

- ESC/Java: extended static checking, uses intermediate language (guarded commands)
- JACK: backwards propagation of invariants, support for native methods, support for bytecode (BML)
- Krakatoa/Why: uses intermediate language, also supports C (Caduceus)
- Jive: uses Hoare logic

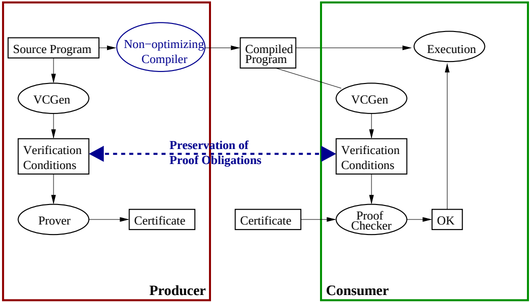
Also relevant: JSR 308, Spec# for C#

Summary

- We have developed:
  - Sound VC generator for sequential fragment of JVM
  - PPO for Java
- Next goal is to provide support for realistic applications:
  - more specification constructs
  - link with JML-based tools
- Other goals
  - certificate generation
  - optimisations
  - concurrency



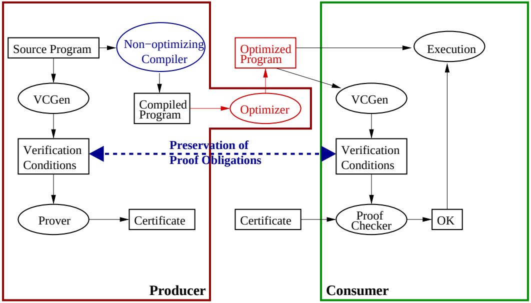
Optimizing Compilers



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)

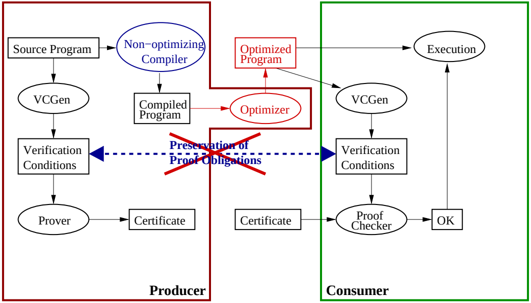
Optimizing Compilers



Proofs obligations might not be preserved

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

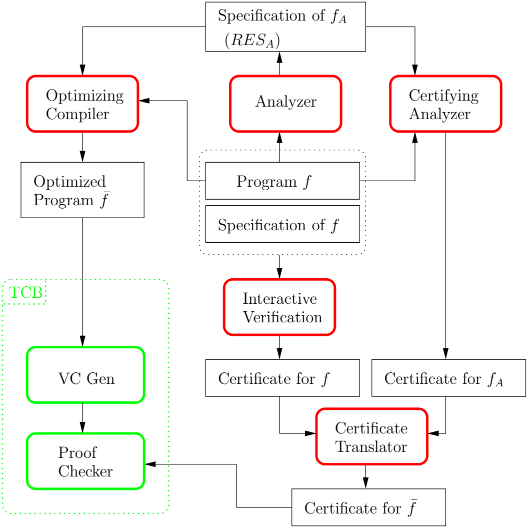
Optimizing Compilers



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;
```

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

```
while(i != n)

  i := c + i

  j := x * i;

endwhile;
```

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

Program + Specification

Motivating example

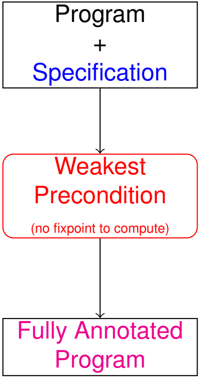
$\{j = 0\}$

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

$\quad \texttt{i} := \texttt{c} + \texttt{i}$

$\quad \texttt{j} := \texttt{x} * \texttt{i};$

$\texttt{endwhile};$
$\{n * b \leq j\}$



Motivating example

$\{j = 0\}$

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

```
i := 0;
```

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

```
x := b + i;
```

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

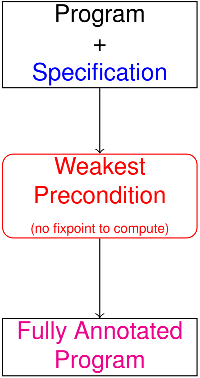
```
while(i != n)

  i := c + i

  j := x * i;

endwhile;
```

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

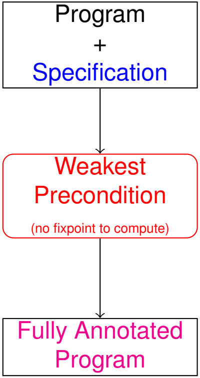


Motivating example

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

`  i := c + i`

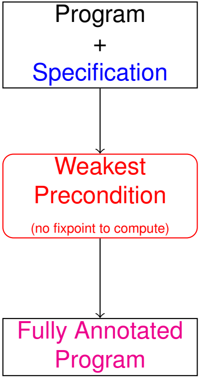
`  j := x * i;`
$\{j = x*i \wedge b \leq x \wedge 0 \leq i\}$
`endwhile;`
$\{n*b \leq j\}$



Motivating example

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

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



Motivating example

$\{j = 0\}$

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

`i := 0;`

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

`x := b + i;`

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

`while(i! = n)`

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

`  i := c + i`

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

`  j := x * i;`

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

`endwhile;`

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

Program
+
Specification

↓

Weakest
Precondition
(no fixpoint to compute)

↓

Fully Annotated
Program

Motivating example

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

Set of Proof Obligations:

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

Constant propagation analysis

$\{j = 0\}$

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

$\texttt{i} := 0;$

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

$(i, 0) \longrightarrow \quad \texttt{x} := \texttt{b} + \texttt{i};$

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

$(x, b) \longrightarrow \quad \texttt{while}(\texttt{i} != \texttt{n})$

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

$(x, b) \longrightarrow \quad \texttt{i} := \texttt{c} + \texttt{i}$

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

$(x, b) \longrightarrow \quad \texttt{j} := \texttt{x} * \texttt{i};$

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

$\texttt{endwhile};$

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

Program transformation

$\{j = 0\}$

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

`i := 0;`

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

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

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

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

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

$(x, b) \longrightarrow$ `i := c + i`

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

$(x, b) \longrightarrow$ `j := x * i;`

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

`endwhile;`

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

Program transformation

$\{j = 0\}$

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

`i := 0;`

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

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

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

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

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

$(x, b) \longrightarrow$ `i := c + i`

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

$(x, b) \longrightarrow$ `j :=` b `* i;`

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

`endwhile;`

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

WP Computation of optimized program

$\{j = 0\}$

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

$\texttt{i} := 0;$

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

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

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

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

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

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

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

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

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

$\texttt{endwhile};$

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

WP Computation of optimized program

$\{j = 0\}$

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

`i := 0;`

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

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

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

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

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

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

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

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

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

`endwhile;`

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

WP Computation of optimized program

$\{j = 0\}$

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

`i := 0;`

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

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

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

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

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

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

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

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

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

`endwhile;`

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

WP Computation of optimized program

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

WP Computation of optimized program

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

Proof Obligations

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

Proof Obligations:

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

Proof Obligations

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

Proof Obligations:

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

Unprovable without knowing $x = b$

Proof Obligations

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

Proof Obligations:

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

   Solution: strengthen annotations
3. $j = x * i \wedge b \leq x \wedge 0 \leq i \wedge i = n \Rightarrow n * b \leq j$

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
$\{\varphi_3\}$		$\{\varphi_3 \wedge \psi_3\}$

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

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

If the analysis is correct,

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

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
$\{\varphi_3\}$		$\{\varphi_3 \wedge \psi_3\}$

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

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

If the analysis is correct,

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

are valid proof obligations.

Strengthening annotations

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

$\mathtt{S}_1$		$\mathtt{S}_1$
$\{\varphi_1\}$		$\{\varphi_1 \wedge \psi_1\}$
$\mathtt{S}_2$		$\mathtt{S}_2$
$\{\varphi_2\}$	$\rightsquigarrow$	$\{\varphi_2 \wedge \psi_2\}$
$\mathtt{S}_3$		$\mathtt{S}_3$
$\{\varphi_3\}$		$\{\varphi_3 \wedge \psi_3\}$

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

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

If the analysis is correct,

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

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
$\{\varphi_3\}$		$\{\varphi_3 \wedge \psi_3\}$

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

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

If the analysis is correct,

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

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

If the analysis is correct,

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

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

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

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

If the analysis is correct,

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

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

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

Certifying analysis for constant propagation

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

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

$$
\begin{array}{ccccccc}
\{\phi_1\} & + & \{\phi_1^A\} & \rightarrow & \{\phi_1 \wedge \phi_1^A\} & & \{\phi'_1 \wedge \phi_1^A\} \\
S_1 & & S_1 & & S_1 & \rightarrow & S_1^O \\
\{\phi_2\} & + & \{\phi_2^A\} & \rightarrow & \{\phi_2 \wedge \phi_2^A\} & & \{\phi'_2 \wedge \phi_2^A\} \\
S_2 & & S_2 & & S_2 & \rightarrow & S_2^O \\
\vdots & + & \vdots & \rightarrow & \vdots & & \vdots \\
S_{n-1} & & S_{n-1} & & S_{n-1} & \rightarrow & S_{n-1}^O \\
\{\phi_n\} & + & \{\phi_n^A\} & \rightarrow & \{\phi_n \wedge \phi_n^A\} & & \{\phi'_n \wedge \phi_n^A\} \\
S_n & & S_n & & S_n & \rightarrow & S_n^O
\end{array}
$$

Translation consists of:

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

$$
\begin{array}{ccccccc}
\{\phi_1\} & + & \{\phi_1^A\} & \to & \{\phi_1 \wedge \phi_1^A\} & & \{\phi'_1 \wedge \phi_1^A\} \\
S_1 & & S_1 & & S_1 & \to & S_1^O \\
\{\phi_2\} & + & \{\phi_2^A\} & \to & \{\phi_2 \wedge \phi_2^A\} & & \{\phi'_2 \wedge \phi_2^A\} \\
S_2 & & S_2 & & S_2 & \to & S_2^O \\
\vdots & + & \vdots & \to & \vdots & & \vdots \\
S_{n-1} & & S_{n-1} & & S_{n-1} & \to & S_{n-1}^O \\
\{\phi_n\} & + & \{\phi_n^A\} & \to & \{\phi_n \wedge \phi_n^A\} & & \{\phi'_n \wedge \phi_n^A\} \\
S_n & & S_n & & S_n & \to & S_n^O
\end{array}
$$

Translation consists of:

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

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

Translation consists of:

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

Certificates

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

Proof algebra

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

Merging certificates

We need to build from the original and analysis certificates:

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

the certificate for the optimized program:

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

by using the gluing lemma

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

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

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

Merging certificates

We need to build from the original and analysis certificates:

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

the certificate for the optimized program:

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

by using the gluing lemma

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

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

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

Merging certificates

We need to build from the original and analysis certificates:

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

the certificate for the optimized program:

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

by using the gluing lemma

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

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

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

Merging certificates

We need to build from the original and analysis certificates:

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

the certificate for the optimized program:

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

by using the gluing lemma

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

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

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

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

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

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

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

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

can be derived from the original one:

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

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

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

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

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

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

can be derived from the original one:

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

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

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

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

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

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

can be derived from the original one:

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

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

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

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

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

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

can be derived from the original one:

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

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

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

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

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

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

can be derived from the original one:

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

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

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

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

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

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

can be derived from the original one:

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

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

$\{\varphi_1\}$	$\{\mathrm{true}\}$	$\{\varphi_1 \wedge \mathrm{true}\}$
`x := 5;`	`x := 5;`	`x := 5;`
$\{\varphi_2\}$	$\{x = 5\}$	$\{\varphi_2 \wedge x = 5\}$
`y := x`	`y := x`	`y := 5`
$\{\varphi_3\}$	$\{x = 5\}$	$\{\varphi_3 \wedge x = 5\}$

Original PO's:

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

Analysis PO's :

- $\mathrm{true} \Rightarrow 5 = 5$
- $x = 5 \Rightarrow x = 5$

Final PO's:

- $\varphi_1 \wedge \mathrm{true} \Rightarrow \varphi_2[{}^5\!/\!_x] \wedge 5 = 5$
- $\varphi_2 \wedge x = 5 \Rightarrow \varphi_3[{}^5\!/\!_y] \wedge x = 5$

Original and new proof obligations differ

With the gluing lemma ($\forall\phi, e.x = e \wedge \phi[{}^x\!/\!_y] \Rightarrow \phi[{}^e\!/\!_y]$), the original PO entails the new PO

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

$\{\varphi_1\}$	$\{\mathrm{true}\}$	$\{\varphi_1 \wedge \mathrm{true}\}$
`x := 5;`	`x := 5;`	`x := 5;`
$\{\varphi_2\}$	$\{x = 5\}$	$\{\varphi_2 \wedge x = 5\}$
`y := x`	`y := x`	`y := 5`
$\{\varphi_3\}$	$\{x = 5\}$	$\{\varphi_3 \wedge x = 5\}$

Original PO's:

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

Analysis PO's :

- $\mathrm{true} \Rightarrow 5 = 5$
- $x = 5 \Rightarrow x = 5$

Final PO's:

- $\varphi_1 \wedge \mathrm{true} \Rightarrow \varphi_2[^5/_x] \wedge 5 = 5$
- $\varphi_2 \wedge x = 5 \Rightarrow \varphi_3[^5/_y] \wedge x = 5$

Original and new proof obligations differ

With the gluing lemma ($\forall\phi, e.x = e \wedge \phi[^x/_y] \Rightarrow \phi[^e/_y]$), the original PO entails the new PO

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

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

$\{\mathrm{true}\}$
`x := 5;`
$\{x = 5\}$
`y := x`
$\{x = 5\}$

$\{\varphi_1 \wedge \mathrm{true}\}$
`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 :

- $\mathrm{true} \Rightarrow 5 = 5$
- $x = 5 \Rightarrow x = 5$

Final PO's:

- $\varphi_1 \wedge \mathrm{true} \Rightarrow \varphi_2[{}^5\!/\!_x] \wedge 5 = 5$
- $\varphi_2 \wedge x = 5 \Rightarrow \varphi_3[{}^5\!/\!_y] \wedge x = 5$

Original and new proof obligations differ

With the gluing lemma $(\forall\phi, e.x = e \wedge \phi[{}^x\!/\!_y] \Rightarrow \phi[{}^e\!/\!_y])$, the original PO entails the new PO

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

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

$\{\mathrm{true}\}$
`x := 5;`
$\{x = 5\}$
`y := x`
$\{x = 5\}$

$\{\varphi_1 \wedge \mathrm{true}\}$
`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 :

- $\mathrm{true} \Rightarrow 5 = 5$
- $x = 5 \Rightarrow x = 5$

Final PO's:

- $\varphi_1 \wedge \mathrm{true} \Rightarrow \varphi_2[^5/_x] \wedge 5 = 5$
- $\varphi_2 \wedge x = 5 \Rightarrow \varphi_3[^5/_y] \wedge x = 5$

Original and new proof obligations differ

With the gluing lemma ($\forall\phi, e.x = e \wedge \phi[^x/_y] \Rightarrow \phi[^e/_y]$), the original PO entails the new PO

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

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

$\{\text{true}\}$
`x := 5;`
$\{x = 5\}$
`y := x`
$\{x = 5\}$

$\{\varphi_1 \wedge \text{true}\}$
`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 :

- $\text{true} \Rightarrow 5 = 5$
- $x = 5 \Rightarrow x = 5$

Final PO's:

- $\varphi_1 \wedge \text{true} \Rightarrow \varphi_2[{}^5\!/\!_x] \wedge 5 = 5$
- $\varphi_2 \wedge x = 5 \Rightarrow \varphi_3[{}^5\!/\!_y] \wedge x = 5$

Original and new proof obligations differ

With the gluing lemma $(\forall\phi, e.x = e \wedge \phi[{}^x\!/\!_y] \Rightarrow \phi[{}^e\!/\!_y])$, the original PO entails the new PO

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

Optimizing compilers try to avoid duplication of computations.

If x already stores the value of e, then

$$\begin{array}{ccc} \texttt{...} & & \texttt{...} \\ \texttt{y := e} & \xrightarrow{x=e} & \texttt{y := x} \\ \texttt{...} & & \texttt{...} \end{array}$$

The gluing lemma states in this case:

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

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

can be derived from the original one:

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

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

Optimizing compilers try to avoid duplication of computations.

If x already stores the value of e, then

$$\begin{array}{lcl} \texttt{...} & & \texttt{...} \\ \texttt{y := e} & \xrightarrow{x=e} & \texttt{y := x} \\ \texttt{...} & & \texttt{...} \end{array}$$

The gluing lemma states in this case:

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

$$\mathrm{wp}(y := x, \varphi) \quad (= \varphi[{}^{x}\!/\!{}_{y}])$$

can be derived from the original one:

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

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

Optimizing compilers try to avoid duplication of computations.

If x already stores the value of e, then

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

The gluing lemma states in this case:

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

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

can be derived from the original one:

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

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

Optimizing compilers try to avoid duplication of computations.

If x already stores the value of e, then

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

The gluing lemma states in this case:

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

the weakest precondition applied to the transformed instruction

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

can be derived from the original one:

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

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

Optimizing compilers try to avoid duplication of computations.

If x already stores the value of e, then

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

The gluing lemma states in this case:

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

the weakest precondition applied to the transformed instruction

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

can be derived from the original one:

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

Optimizing compilers try to avoid duplication of computations.

If x already stores the value of e, then

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

The gluing lemma states in this case:

> Under the hypothesis that the result of the analysis is valid $x = e$
>
> the weakest precondition applied to the transformed instruction
>
> $$\mathrm{wp}(y := x, \varphi) \quad (\equiv \varphi[{}^{x}\!/\!{}_{y}])$$
>
> can be derived from the original one:
>
> $$\mathrm{wp}(y := e, \varphi) \quad (\equiv \varphi[{}^{e}\!/\!{}_{y}])$$

Applicability of method

- Certificate translators for constant propagation, common sub-expression elimination. . .
- However, representing the result of the analysis as assertions is only possible for analyses that focus on *state properties*
- Several program analyses focus on *execution traces*, e.g. dead variable elimination. We have developed ad-hoc techniques for those.

Applicability of method

- Certificate translators for constant propagation, common sub-expression elimination. . .
- However, representing the result of the analysis as assertions is only possible for analyses that focus on *state properties*
- Several program analyses focus on *execution traces*, e.g. dead variable elimination. We have developed ad-hoc techniques for those.

Applicability of method

- Certificate translators for constant propagation, common sub-expression elimination. . .
- However, representing the result of the analysis as assertions is only possible for analyses that focus on *state properties*
- Several program analyses focus on *execution traces*, e.g. dead variable elimination. We have developed ad-hoc techniques for those.

Ongoing work

- Prototype
  - Source language: imperative language with functions and arrays
  - Target language: RTL with functions and arrays
  - Compiler: performs common optimizations
  - Verification condition generator: interfaced with Coq
- Certificates
  - Size of certificates does not seem to explode, provided one does local normalization of certificates
  - Type checking modulo selected isomorphisms of types could limit certificate growth
- Abstract framework to prove the existence and correctness of certicate translators
- More expressive languages, more complete compilation chains

Concluding remarks

Contents

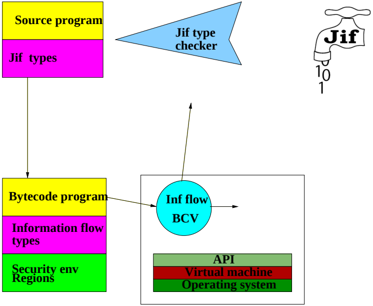
Two verification methods for bytecode and their relation to verification methods for source code

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

Contents

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

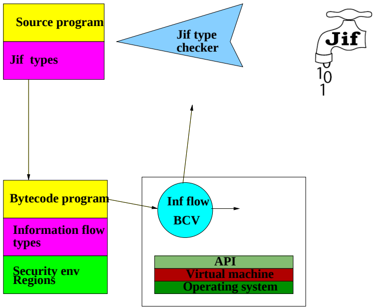
- Type system for information flow based confidentiality policies
- Verification condition generator for logical specifications

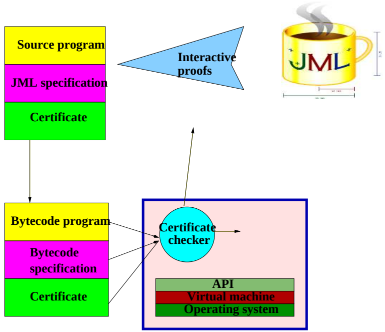


Contents

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)

Many challenges ahead e.g.:

- proof carrying code in distributed setting (result certification)
- combination of language-based and cryptographic-based security

Mobius

http://mobius.inria.fr