Language-based methods for software security

Gilles Barthe

IMDEA Software, Madrid, Spain

Part 1
Mobile code is ubiquitous: large distributed networks of JVM devices
- aimed at providing a global and uniform access to services
- provide support to untrusted mobile code

Security is a central concern: untrusted code may
- use too many resources
  - CPU, memory . . .
- perform unauthorized actions
  - open sockets
- be hostile towards other applications
  - access, manipulate or reveal sensitive data
- crash the system
  - destruction/corruption of files
Certificates

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

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

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

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

Logic-based PCC
- Original scenario
- Application to type safety and memory safety

- Logic-based methods for software security
the program is annotated (loop invariants, function specifications),

- the VCGen computes a logic formula $\phi$ that if true guarantees the program security,

- the certifying prover computes a proof object $\pi$ which establishes the validity of $\phi$,

- the consumer rebuilds the formula $\phi$ and checks that $\pi$ is a valid proof of $\phi$. 
the program is annotated (loop invariants, function specifications),

the VCGen computes a logic formula $\phi$ that if true guarantees the program security,

the certifying prover computes a proof object $\pi$ which establishes the validity of $\phi$,

the consumer rebuilds the formula $\phi$ and checks that $\pi$ is a valid proof of $\phi$. 
the program is annotated (loop invariants, function specifications),

the VCGen computes a logic formula $\phi$ that if true guarantees the program security,

the certifying prover computes a proof object $\pi$ which establishes the validity of $\phi$,

the consumer rebuilds the formula $\phi$ and checks that $\pi$ is a valid proof of $\phi$. 
the program is annotated (loop invariants, function specifications),

the VCGen computes a logic formula $\phi$ that if true guarantees the program security,

the certifying prover computes a proof object $\pi$ which establishes the validity of $\phi$,

the consumer rebuilds the formula $\phi$ and checks that $\pi$ is a valid proof of $\phi$. 

the program is annotated (loop invariants, function specifications),

the VCGen computes a logic formula $\phi$ that if true guarantees the program security,

the certifying prover computes a *proof object* $\pi$ which establishes the validity of $\phi$,

the consumer rebuilds the formula $\phi$ and checks that $\pi$ is a valid proof of $\phi$. 
Certifying prover

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

Touchstone’s certifying prover includes
- congruence closure and linear arithmetic decision procedures
- with a Nelson-Oppen architecture for cooperating decision procedures
the transmitted program is the result of the compilation of a source program written in a type-safe language
the role of the certifying compiler is
- to check type-safety of the source program
- to generate corresponding annotations in the machine code to help the VCGen
The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?

- the proof checker
- the VCGen
- the Execution platform

**Certifying compiler**

Source code → Machine code → Annotations → VCGen

VCGen → Certifying prover → Proof checker

Execution platform
Trusted Computing Base (TCB)

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

What is the PCC TCB?
- the proof checker
- the VCGen
- the Execution platform
The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?
- the proof checker
- the VCGen
- the Execution platform

You don’t need to trust ...
The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?
- the proof checker
- the VCGen
- the Execution platform

You don’t need to trust the compiler ...
The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?
- the proof checker
- the VCGen
- the Execution platform

You don’t need to trust the compiler, the annotations ...
The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?
- the proof checker
- the VCGen
- the Execution platform

You don’t need to trust the compiler, the annotations, the prover ...
The TCB of a program is the set of components that must be trusted to ensure the soundness of the program. Any bug in the others components will never affect the soundness.

What is the PCC TCB?
- the proof checker
- the VCGen
- the Execution platform

You don’t need to trust the compiler, the annotations, the prover, the proof ...
Other instances of PCC

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

The size of the TCB can be reduced
- by relying on simpler checkers
- by removing the VCGen: *Foundational Proof-Carrying Code*
- by certifying the VCGen in a proof assistant

---

\(^1\)G.C. Necula and R.R. Schneck. *A Sound Framework for Untrusted Verification-Condition Generators*. LICS’03
Simpler checkers?

Proof

\[
\begin{align*}
\hat{\delta}[P] (\text{Post}[\text{if } B \text{ then } S_t \text{ else } S_f \text{ fi}]) &= \\
&= \{\text{def. (110) of } \hat{\delta}[P]\} \\
&= \hat{\delta}[P] \circ \text{Post}[\text{if } B \text{ then } S_t \text{ else } S_f \text{ fi}] \circ \hat{\gamma}[P] \\
&= \{\text{def. (103) of Post}\} \\
&= \hat{\delta}[P] \circ \text{post}[r^*[\text{if } B \text{ then } S_t \text{ else } S_f \text{ fi}]] \circ \hat{\gamma}[P] \\
&= \{\text{big step operational semantics (93)}\} \\
&= \{\text{Galois connection (98) so that post preserves joins}\} \\
&= \hat{\delta}[P] \circ \text{post}[(1_{\Sigma[P]} \cup \tau^B) \circ r^*[S_t] \circ (1_{\Sigma[P]} \cup \tau^I) \circ (1_{\Sigma[P]} \cup \tau^I)] \circ \hat{\gamma}[P] \\
&= \{\text{Galois connection (106) so that } \hat{\delta}[P] \text{ preserves joins}\} \\
&= \{\text{lemma (5.3) and similar one for the else branch}\} \\
&\lambda J \cdot \text{let } J' = \lambda l \in \text{in}_P[P](U = \text{at}_P[S_t] \land J_{\text{at}_P[S_t]} \cup \text{Abexp}[B](J_t) \land J_t) \text{ in } \\
&\quad \text{let } J'' = \text{APost}_1[S_t](J') \text{ in } \\
&\quad \lambda l \in \text{in}_P[P](U = l' \star J'' \land J_{\text{after}_P[S_t]}(l) \star J''_I) \\
&\quad \cup \\
&\quad \lambda l \in \text{in}_P[P](U = l' \star J'' \land J_{\text{after}_P[S_t]}(l) \star J''_I) \\
&= \{\text{by grouping similar terms}\} \\
&\lambda J \cdot \text{let } J' = \lambda l \in \text{in}_P[P](U = \text{at}_P[S_t] \land J_{\text{at}_P[S_t]} \cup \text{Abexp}[B](J_t) \land J_t) \\
&\text{ and } J'' = \lambda l \in \text{in}_P[P](U = \text{at}_P[S_t] \land J_{\text{at}_P[S_t]} \cup \text{Abexp}[B](J_t) \land J_t) \text{ in } \\
&\quad \text{let } J'' = \text{APost}_1[S_t](J') \text{ in } \\
&\quad \lambda l \in \text{in}_P[P](U = l' \star J'' \land J_{\text{after}_P[S_t]}(l) \star J''_I) \\
&= \{\text{by locality (113) and labelling scheme (59) so that in particular } J''_I = J''_I = J''_I = J''_I = J''_I \text{ and APost}[S_t] \text{ and APost}[S_t] \text{ do not interfere}\} \\
\end{align*}
\]
Proof

\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
\[ J = J' \]
Implementation

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

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

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

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

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

- Dataflow analysis ensures that values are manipulated with correct types, methods are applied to correct arguments, no stack underflows and overflows.

- Preceded by a structural analysis that ensures that the code is well-formed and methods, names, and classes exist.

- and that jumps remain with code!

- In 2004, Godwiak exploited failure of BCV to verify targets of jumps to launch attacks on Nokia phones

- No verifier for a real language is really simple!
Really simple checkers?

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

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

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

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

Executions of program \( p \) are safe.

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

Executions of program $p$ are safe.

- Proof proceeds by showing that safety is an invariant of execution, under assumptions given for $p$
- depends on the definition of execution.
  - For the JVM: a 400 pages book!

TCB of Foundational PCC:
- the proof checker (as before)
- the formal definition of the language semantics
- the formal definition of the policy

This is also a large TCB

Still better to have 2,000 lines of formal definitions than with 20,000 lines of C code!
Theorem

Executions of program p are safe.

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

Theorem

Executions of program $p$ are safe.

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

*Executions of program p are safe.*

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

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

Reflection

Use computations instead of deductions!
- A predicate $P : T \rightarrow \text{Prop}$
- A decision procedure $f : T \rightarrow \text{bool}$
- A correctness lemma $C : \forall x : T. f \ x = \text{true} \rightarrow P \ x$

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

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

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

(same as FPCC)
TCB of certified PCC

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

(same as FPCC)
TCB of certified PCC

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

(same as FPCC)
In standard PCC

1. If the VCGen is proved correct
   - the proof checker
   - the formal definition of the language semantics
   - the formal definition of the policy

(same as FPCC)
In standard PCC

1. If the VCGen is proved correct
   - the proof checker
   - the formal definition of the language semantics
   - the formal definition of the policy

(same as FPCC)
Using executable checkers

Producer

Consumer

Gilles Barthe

Language-based methods for software security
Using executable checkers

Producer

Consumer

Coq kernel

semantics + policy

+ policy

Gilles Barthe

Language-based methods for software security
Using executable checkers

Producer

Certified verifier

(certified verifier) (Coq file)

Consumer

Certified verifier

Semantics + policy

Coq kernel

Certificate verifier

Checks certified solution

Gilles Barthe
Language-based methods for software security
Using executable checkers

Producer

certified
verifier

(coq file)

certified
verifier

Consumer

certified
verifier

(semantics +
policy)

Coq kernel

solution

program

Safe?

checks certified solution

certificate
verifier

Language-based methods for software security
Using executable checkers

Producer

- certified verifier
- untrusted solver

Consumer

- certified verifier
  (Coq file)
- semantics + policy
- Coq kernel
- certificate verifier

Solution flows:

- untrusted solver computes (certified) solution
- solution flows to certified verifier
- Coq kernel checks certified solution
- program flows to certificate verifier
- checks certified solution
- Safe?
Using executable checkers

Producer

- certified verifier
- untrusted solver
- untrusted compressor

Consumer

- certified verifier
- semantics + policy
- Coq kernel
- certificate verifier

Solution
- computes (certified) solution
- checks certified solution
- Safe?

Program
Application scenario: PCC with trusted intermediaries

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

Producer
- certified verifier
- untrusted solver
- untrusted compressor

Consumer and verifier
- certified verifier
- (Coq file)
- semantics + policy
- Coq kernel (+ Coq extraction)
- (extracted) certificate

Solution
- computes (certified) solution
- inclusion certificates
- solution

Program
- checks certified solution
- Safe?
Application scenario: retail PCC

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

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

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

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

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

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

- Type system for information flow based confidentiality policies
- Verification condition generator for logical specifications
Proof assistants based on type theory

Type theory is a language for:
- defining mathematical objects (including data structures, algorithms, and mathematical theories)
- performing computations on and with these objects
- reasoning about these objects

It is a foundational language that underlies:
- proof assistants (inc. Coq, Epigram, Agda)
- programming languages (inc. Cayenne, DML).
Proof assistants

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

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

Proof objects as certificates

- Completed proofs are represented by proof objects that can easily be checked by a proof-checker.
- Proof checker is small.
Sample applications (many more)

- Programming languages
  - Programming language semantics
  - Program transformations: compilers, partial evaluators, normalizers
  - Program verification: type systems, Hoare logics, verification condition generators,

- Operating systems

- Cryptographic protocols and algorithms
  - Dolev-Yao model (perfect cryptography assumption)
  - Computational model

- Mathematics and logic:
  - Galois theory, category theory, real numbers, polynomials, computer algebra systems, geometry, group theory, etc.
  - 4-colors theorem
  - Type theory
Type theory is a programming language for writing algorithms.
- But all functions are total and terminating, so that convertibility is decidable.

Type theory is a language for proofs, via the Curry-Howard isomorphism:
- Propositions = Types
- Proofs = Terms
- Proof-Checking = Type-Checking

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

- Judgements
  \[ x_1 : A_1, \ldots, x_n : A_n \vdash M : B \]

- Typing rules
  \[
  \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \quad \frac{\Gamma \vdash M : A \rightarrow B}{\Gamma \vdash M \, N : B} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A. M : A \rightarrow B}
  \]

- Evaluation: computing the application a function to an argument
  \[
  (\lambda x : A. M) \, N \rightarrow_{\beta} M[x := N]
  \]

- The result of computation is unique
  \[
  M =_{\beta} N \Rightarrow M \downarrow_{\beta} N
  \]

- Evaluation preserves typing
- Type-Checking: it is decidable whether \( \Gamma \vdash M : A \).
- Type-Inference: there exists a partial function \( \inf \) s.t.
  \[
  \Gamma \vdash M : A \iff \Gamma \vdash M : (\inf(\Gamma, M)) \land (\inf(\Gamma, M)) = A
  \]
A Language for Proofs

Minimal Intuitionistic Logic

- **Formulae:**
  \[ \mathcal{F} = \mathcal{X} \]
  \[ \mid \mathcal{F} \to \mathcal{F} \]

- **Judgements**
  \[ A_1, \ldots, A_n \vdash B \]

- **Derivation rules**

  \[ \text{If } \Gamma \vdash A \text{ then } \Gamma \vdash A \]

  \[ \text{If } \Gamma \vdash A \text{ then } \Gamma \vdash M : A \text{ for some } M \]

  \[ (\text{A tight correspondence between derivation trees and } \lambda\text{-terms, and between proof normalization and } \beta\text{-reduction}) \]

  \[ \text{In a proof assistant } M \text{ is often built backwards.} \]
BHK Interpretation

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

\[
\begin{align*}
N &: \text{Type}, O : N, P : N \rightarrow \text{Prop} \\
\vdash \lambda x : (P O). x : (P O) &\rightarrow P((\lambda z : N. z) O)
\end{align*}
\]
Typing dependent types: Calculus of Constructions

\[ \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad (s_1, s_2) \in \mathcal{R} \]

\[ \Gamma \vdash (\Pi x : A. B) : s_2 \]

\[ \Gamma \vdash F : (\Pi x : A. B) \quad \Gamma \vdash a : A \]

\[ \Gamma \vdash F a : B\{x := a\} \]

\[ \Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A. B) : s \]

\[ \Gamma \vdash \lambda x : A. b : \Pi x : A. B \]

\[ \Gamma \vdash A : B \quad \Gamma \vdash B' : s \]

\[ B =_{\beta} B' \]

Rules

- (Prop, Prop)
- implication
- (Type, Type)
- generalized function space
- (Type, Prop)
- universal quantification
- (Prop, Type)
- precondition, etc
Inductive definitions provide mechanisms to define data structures, to define recursive functions and to reason about inhabitants of data structures

- recursors/case-expressions and guarded fixpoints/pattern matching
- induction principles

Encode a rich class of structures:

- algebraic types: booleans, binary natural numbers, integers, etc
- parameterized types: lists, trees, etc
- inductive families and relations: vectors, accessibility relations (to define functions by well-founded recursion), transition systems, etc.

Extensively used in the formalization of mathematics, programming languages, cryptographic algorithms, in reflexive tactics, etc.
Typing rules for natural numbers

\[
\begin{align*}
\vdash \text{Nat} : s & \quad \vdash 0 : \text{Nat} \\
\Gamma \vdash n : \text{Nat} & \quad \Gamma \vdash S \, n : \text{Nat} \\
& \quad \vdash n : \text{Nat} \quad \Gamma \vdash f_0 : A \quad \Gamma \vdash f_s : \text{Nat} \rightarrow A \\
& \quad \quad \quad \Gamma \vdash \text{case} \ n \ \text{of} \{0 \Rightarrow f_0 \ | \ s \Rightarrow f_s\} : A \\
& \quad \quad \quad \quad \vdash n : \text{Nat} \quad \Gamma \vdash P : \text{Nat} \rightarrow s \\
& \quad \quad \quad \quad \Gamma \vdash f_0 : P \, 0 \quad \Gamma \vdash f_s : \Pi n : \text{Nat}. \ P \ (S \ n) \\
& \quad \quad \quad \quad \quad \Gamma \vdash \text{case} \ n \ \text{of} \{0 \Rightarrow f_0 \ | \ s \Rightarrow f_s\} : P \ n \\
& \quad \quad \quad \quad \quad \quad \Gamma, f : \text{Nat} \rightarrow A \vdash e : \text{Nat} \rightarrow A \\
& \quad \quad \quad \quad \quad \quad \quad \Gamma \vdash \text{letrec} \ f = e : \text{Nat} \rightarrow A
\end{align*}
\]
Case expressions and fixpoints: reduction rules

\[
\text{case } 0 \text{ of } \{ 0 \Rightarrow e_0 \mid s \Rightarrow e_s \} \rightarrow e_0 \\
\text{case } (s\ n) \text{ of } \{ 0 \Rightarrow e_0 \mid s \Rightarrow e_s \} \rightarrow e_s\ n \\
\ (\text{letrec } f = e)\ n \rightarrow e[f \leftarrow (\text{letrec } f = e)]\ n
\]

To ensure termination

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

Not sufficient to impose restrictions on fixpoint definitions. Must also guarantee inductive definitions are well-formed.
Example: formalizing semantics of expressions

\[ a \in \text{AExp} \quad b \in \text{BExp} \quad c \in \text{Comm} \]

\[
\begin{align*}
    a &:= n \\
    &\mid x \\
    &\mid a_1 + a_2 \\
    &\mid a_1 - a_2 \\
    &\mid a_1 \ast a_2 \\

    b &:= \text{true} \\
    &\mid \text{false} \\
    &\mid a_1 = a_2 \\
    &\mid a_1 < a_2 \\
    &\mid \text{not } b \\

    c &:= \text{skip} \\
    &\mid x := a \\
    &\mid c_1; c_2 \\
    &\mid \text{if } b \text{ then } c_1 \text{ else } c_2 \\
    &\mid \text{while } b \text{ do } c \\
    &\mid b_1 \text{ and } b_2
\end{align*}
\]
Shallow embedding

- Expressions have type \( \text{mem} \rightarrow \text{Nat} \)
- Memories have type \( \text{mem} = \text{loc} \rightarrow \text{Nat} \)

\[
\text{Num}[v: \text{Nat}] = \lambda s: \text{mem}. \, v
\]
\[
\text{Loc}[v: \text{loc}] = \lambda s: \text{mem}. \, s \, v
\]
\[
\text{Plus}[e_1, e_2: \text{Exp}] = \lambda s: \text{mem}. \, (e_1 \, s) + (e_2 \, s)
\]
\[
\text{Minus}[e_1, e_2: \text{Exp}] = \lambda s: \text{mem}. \, (e_1 \, s) - (e_2 \, s)
\]
\[
\text{Mult}[e_1, e_2: \text{Exp}] = \lambda s: \text{mem}. \, (e_1 \, s) \ast (e_2 \, s)
\]

\[
x, y: \text{Exp} \vdash \text{Plus} \, x \, (\text{Minus} \, y \, (\text{Num} \, 3)) : \text{Exp}
\]

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

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

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

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

Inductive `com` : Set :=
  Skip: com
  | Assign: loc -> aExp -> com
  | Scolon: com -> com -> com
  | IfThenElse: bExp -> com -> com -> com
  | WhileDo: bExp -> com -> com.
Memory \( \text{mem} = \text{loc} \rightarrow \text{Nat} \)

Evaluation relation \( \langle a, \sigma \rangle \rightarrow^a n \), i.e. \( \rightarrow^a \subseteq A\text{Exp} \times \Sigma \times \mathbb{N} \)

Evaluation rules

\[
\begin{align*}
\langle n, \sigma \rangle & \rightarrow^a n \\
\langle x, \sigma \rangle & \rightarrow^a \sigma(x) \\
\langle a_1, \sigma \rangle & \rightarrow^a n_1 \\
\langle a_2, \sigma \rangle & \rightarrow^a n_2 \\
\langle a_1 + a_2, \sigma \rangle & \rightarrow^a n_1 + n_2
\end{align*}
\]

Inductive evalaExp_ind : aExp -> memory -> nat -> Prop :=
  eval_Loc: forall (v:locs)(n:nat)(s : memory),
          (lookup s v)=n -> (evalaExp_ind (Loc v) s n)
  | eval_Num: forall (n : nat) (s : memory),
     (evalaExp_ind (Num n) s n)
  | eval_Plus: forall (a0, a1 : aExp) (n0, n1, n : nat) (s : memory),
           (evalaExp_ind a0 s n0) ->
           (evalaExp_ind a1 s n1) ->
           n = (plus n0 n1) -> (evalaExp_ind (Plus a0 a1) s n)
  ...

Gilles Barthe
Language-based methods for software security
Fixpoint evalaExp_rec [a : aExp] : memory -> nat :=
fun (s : memory) =>
  match a with
  (Loc v) => (lookup s v)
| (Num n) => n
| (Plus a1 a2) => (plus (evalaExp_rec a1 s) (evalaExp_rec a2 s))
| ...
end.

Possible difficulties with functional semantics

- Determinacy
- Partiality
- Termination

For commands:

- Small-step semantics is possible to define but
  - many undefined cases to handle
  - still harder to reason about than inductive semantics

- Big-step semantics is hard (requires well-founded recursion)
Certifying type-based methods

- Bytecode verification
- Abstraction-carrying code
- Non-interference
Bytecode verification aims to contribute to safe execution of programs by enforcing:

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

But well-typed programs do not go wrong

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

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

Informally

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

\[ \vdash \text{pop} : (rt, \alpha :: s) \Rightarrow (rt, s) \quad \nvdash \text{pop} : (rt, s) \Rightarrow (rt, s) \]

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

\[
\begin{align*}
P[i] = \text{ins} & \quad P[i] = \text{ins} \\
\frac{i \vdash \text{lv}, \text{st} \Rightarrow \text{lv}', \text{st}'}{i \vdash \text{lv}, \text{st} \Rightarrow}
\end{align*}
\]

- Program \( P : \tau \) is type-safe if there exists \( S : P \rightarrow RT \times T^* \) s.t.

  - \( S_1 = (rt_1, \epsilon) \)
  - for all \( i, j \in P \)
    - \( i \mapsto j \Rightarrow \exists \sigma. i \vdash S_i \Rightarrow \sigma \sqsubseteq S_j; \)
    - \( i \mapsto \Rightarrow \exists \tau'. i \vdash S_i \Rightarrow \tau' \sqsubseteq \tau \)

where \( \sqsubseteq \) is inherited from JVM types
Bytecode verification: consequences

**Programs do not go wrong**

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

- $P[i] = \text{return}$ then the return value has type $\tau$
- $s \leadsto s'$ and $s'$ is type-correct w.r.t. $S_i'$
  (where $i = pc(s)$ and $i' = pc(s')$)

**Run-time type checking is redundant**

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

Goal is to exhibit $S$.

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

Fixpoint computation!
Lightweight bytecode verification

Provide types of junction points

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

Provide types of junction points

- Entry point and junction points are typed
  - the entry point of the program is typed with the empty stack

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

One pass verification, sound and complete wrt bytecode verification
- A puzzle with 8 pieces,
- Each piece interacts with its neighbors
- a Coq formalisation of the JVM
- the basis for certified PCC

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

**Initial requirements**

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

Bicolano should be

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

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

Bicolano should be

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

Bicolano should be

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

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

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

- **Binary operations on int**: `ibinop op`  
  
  (iadd, iand, idiv, imul, ior, irem, ishl, ishr, isub, iushr, ixor)

- **Tests on int value**: `if0 comp`  
  
  (ifeq, ifne, iflt, ifle, ifgt, ifge)

- **Push numerical constants on the operand stack**: `const t c`  
  
  (bipush, iconst_<i>, ldc, sipush)

- **load value from local variables**: `aload, iload`

- **load value from array**: `aaload, baload, iaload, saload`

- **similar instructions to store values...**
Wellformedness properties on programs

Some examples

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

Coq packaging:

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

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

Proof on wellformed programs:

```coq
forall (p:Program), well_formed_program p -> ...
```
Verified bytecode verification

Example: JVM states

\[ \langle \langle h, \langle m, pc, l, v :: s \rangle, sf \rangle \rangle \]
Inductive value : Set :=
    | Int (v:Z) (* Numeric value *)
    | NULL (* reference *)
    | UNDEF (* default value *).

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

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

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

(* Transfer function between operand stack and local variables *)
Parameter stack2localvar : OperandStack -> nat -> LocalVar.t.
Formalization of JVM states

Heap

Module Type HEAP.
   Parameter t : Type.

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

Inductive LocationType : Set :=
   | LocationObject : ClassName -> LocationType
   | LocationArray : Int -> type -> LocationType.

(** (LocationArray length element type) *)

Parameter typeof : t -> Location -> option LocationType.

(** typeof h loc = None -> no object, no array allocated at location loc *)

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

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

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

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

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

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

\[ x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq \cdots \]

Inherited from JVM types (extension to finite maps and stacks)
Specific challenges, e.g. interfaces

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

Both I and J are upper bounds for C and D, but they are incomparable.
- Each type represents a property on concrete values
- This correspondence is formalised by the relation value : type (that respects subtyping)
Verified bytecode verification

Operational semantics $\leadsto$ between states

$$P[(m, pc)] = \text{push } c$$

$$\langle h, \langle m, pc, l, s \rangle, sf \rangle \leadsto \langle h, \langle m, pc + 1, l, c :: s \rangle, sf \rangle$$

$$P[(m, pc)] = \text{invokevirtual } m_id$$

$$m' = \text{methodLookup}(m_id, h(loc))$$

$$V = v_1 :: \cdots :: v_{\text{nbArguments}}(m_id)$$

$$\langle h, \langle m, pc, l, \text{loc :: } V :: s \rangle, sf \rangle \leadsto \langle h, \langle m', 1, V, \varepsilon \rangle, \langle m, pc, l, s \rangle :: sf \rangle$$
| const_step_ok : for all h m pc pc’ s l sf t z,
| instructionAt m pc = Some (Const t z) →
| next m pc = Some pc’ →
| ( (t=BYTE /\ −2^7 <= z < 2^7)
| \/ (t=SHORT /\ −2^15 <= z < 2^15)
| \/ (t=INT /\ −2^31 <= z < 2^31) ) →
| step p (St h (Fr m pc s l) sf) (St h (Fr m pc’ (Num (I (Int.const z)):s) l) sf)

| invokevirtual_step_ok : for all h m pc s l sf mid cn M args loc cl bM fnew,
| instructionAt m pc = Some (Invokevirtual (cn,mid)) →
| lookup p cn mid (pair cl M) →
| Heap.dtypeof h loc = Some (Heap.LocationObject cn) →
| length args = length (METHODSIGNATURE.parameters mid) →
| METHOD.body M = Some bM →
| fnew = (Fr M
| \quad (BYTECODEMETHOD.firstAddress bM)
| \quad OperandStack.empty
| \quad (stack2localvar (args++(Ref loc):s) (1+(length args)))) ) →
| step p (St h (Fr m pc (args++(Ref loc):s) l) sf) (St h fnew ((Fr m pc s l):sf))
Small step semantics

Two kinds of state:

- **normal state**:
  \[(St \ h \ (Fr \ m \ pc \ s) \ sf)\]

- **exception state (not yet caught)**
  \[(StE \ h \ (FrE \ m \ pc \ loc \ l) \ sf)\]

The small step semantics is defined with a relation between state

\[\text{step (p:Program) : State \rightarrow State \rightarrow Prop}\]
Small step semantics

Four cases

1. normal $\rightarrow$ normal
2. normal $\rightarrow$ exception
3. exception $\rightarrow$ normal
4. exception $\rightarrow$ exception
Small step semantics

Four cases

1. **normal → normal**

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

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

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

2. **normal → exception**

3. **exception → normal**

4. **exception → exception**
Small step semantics
Four cases

1. normal → normal
2. normal → exception

putfield\_step\_NullPointerException :
\(\forall h \ \text{pc} \ s \ l \ sf \ f\ v \ h' \ loc',\)

\[\text{instructionAt } \text{pc} = \text{Some}(\text{Putfield } f) \rightarrow\]
\[\text{Heap.new } h \ p (\text{Heap.LocationObject (javaLang, NullPointerException)})\]
\[= \text{Some}(\text{loc'}, h') \rightarrow\]

\[\text{step } p (\text{St } h (\text{Fr } m \text{pc} (v::Null::s) l) sf)\]
\[= (\text{StE } h' (\text{FrE } m \text{pc loc'} l) sf)\]

3. exception → normal
4. exception → exception
Small step semantics

Four cases

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

\[ \text{exception}\_\text{caught} : \forall h \ m \ pc \ loc \ l \ sf \ bm \ pc', \]

METHOD.body m = Some bm →
lookup.handlers p
(BYTECODEMETHOD.exceptionHandlers bm) h pc loc pc’ →

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

4. exception → exception
Small step semantics
Four cases

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

\[\text{exception\_uncaught : } \forall h m pc \ loc l m' pc' s' l' sf \ \exists bm,\]

\[
\text{METHOD.body } m = \text{Some } bm \rightarrow \\
(\forall pc', \\
\quad \text{lookup Handlers } p \\
\quad (\text{BYTECODEMETHOD.exceptionHandlers } bm) h pc \ loc pc') \rightarrow \\
\text{step } p (\text{StE } h (\text{FrE } m pc \ loc l)) ((\text{Fr } m' pc' s' 1')::sf)) \\
\quad (\text{StE } h (\text{FrE } m' pc' loc l') sf)
\]
The small step semantics is not well suited to prove the correctness of modular verification methods.

Better to reason relative to intermediate semantics with method calls are performed in one-step, or relative to big-step semantics.

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

Still necessary to prove correspondence with the small step semantics.
The big step semantics relies on 4 kinds of elementary steps:

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

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

**Theorem**

Big-step semantics and small-step semantics are equivalent (in some precise mathematical sense based on complete executions)
the type system is specified by transfer rules

\[ \text{tstep } (p:\text{Program}) : \text{tState} \rightarrow \text{tState} \rightarrow \text{Prop}. \]

whose definition is similar to operational semantics

the definition of typability is a direct application of transfer rules

a type is a solution of a fixpoint problem \( F^\#(S) \subseteq (S) \) or equivalently of a constraint system
Sample transfer rules

\[
P[i] = iadd
\]

\[
i \vdash rt, \text{int} :: \text{int} :: st \Rightarrow rt, \text{int} :: st
\]

\[
P[i] = \text{iconst } n \quad |st| + 1 \leq \text{Mstack}
\]

\[
i \vdash rt, st \Rightarrow rt, \text{int} :: st
\]

\[
P[i] = \text{aload } n \quad rt(n) = \tau \quad \tau \prec \text{Object} \quad |st| + 1 \leq \text{Mstack}
\]

\[
rt, st \Rightarrow rt, \tau :: st
\]

\[
P[i] = \text{astore } n \quad \tau \prec \text{Object} \quad 0 \leq n < \text{Mreg}
\]

\[
i \vdash rt, \tau :: st \Rightarrow rt[n \leftarrow \tau], st
\]

\[
P[i] = \text{getfield } C f \tau \quad \tau' \prec C
\]

\[
i \vdash rt, \tau' :: st \Rightarrow rt, \tau :: st
\]

\[
P[i] = \text{putfield } C f \tau \quad \tau_1 \prec \tau \quad \tau_2 \prec C
\]

\[
i \vdash rt, \tau_1 :: \tau_2 :: st \Rightarrow rt, st
\]
If $s \sim s'$ and $s$ is type-correct, then $s'$ is type-correct

- easy proof, but tedious: one proof by instruction
- uses intermediate semantics
- exceptions may be handled separately
From declarative definition of typable program to type checker

- rely on generic construction
- ... but requires discharging hypotheses!
- implement functions for inclusion checking
- provide hypotheses that guarantee termination (for bcv, not lbcv)
Verified bytecode verification

Final results

\[
\begin{align*}
\text{check } P &= \text{ok} \\
S_{\text{init}} &\downarrow S_{\text{final}} \\
S_{\text{init}} &\text{ type – correct}
\end{align*}
\Rightarrow S_{\text{final}} \text{ type – correct}
\]

- progress
- commutation defensive and offensive machine
Types are properties:
  - being an integer
  - being a boolean

More precise types:
  - parity
  - interval
  - etc.

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

#### Parity

- **Abstract properties**
  
  \[
  \text{odd} \quad \text{even}
  \]

- **Least upper bound**
  
  \[
  \text{odd} \sqcup \text{even} = \top
  \]

- **Abstract semantics of addition**
  
  \[
  \begin{align*}
  \text{even} + \text{even} &= \text{even} \\
  \text{odd} + \text{odd} &= \text{even} \\
  \text{even} + \text{odd} &= \text{odd} \\
  x + \top &= \top \\
  x + \bot &= \bot \\
  \ldots &\ldots
  \end{align*}
  \]

#### Intervals

- **Abstract properties**
  
  \[
  [i, j]
  \]
  
  where \( i, j \in \text{int} \sqcup \{+\infty, -\infty\} \)

- **Least upper bound**
  
  \[
  [i, j] \sqcup [i', j'] = [i'', j'']
  \]

  where

  \[
  \begin{align*}
  i'' &= \min(i, i') \\
  j'' &= \max(j, j')
  \end{align*}
  \]

- **Abstract semantics of addition**
  
  \[
  [i, j] + [i', j'] = [i + i', j + j']
  \]
Concrete vs abstract semantics

Program semantics

\[ \{\eta_1, \eta'_1, \eta''_1\} \rightarrow l_1 \]
\[ \{\eta_2, \eta'_2\} \rightarrow l_2 \]
\[ \{\eta_3\} \rightarrow l_3 \quad \{\eta_5'\} \rightarrow l_5 \]
\[ \ldots \]
\[ \{\eta_f, \eta'_f, \eta''_f\} \rightarrow l_f \]
Concrete vs abstract semantics

Program semantics

Abstract representation

\[ \{\eta_1, \eta'_1, \eta''_1\} \]

\[ \{\eta_2, \eta'_2\} \]

\[ \{\eta_3\} \]

\[ \{\eta_5', \eta'_5, \eta''_5\} \]

\[ \{\eta_f, \eta'_f, \eta''_f\} \]

\[ a_1 \]

\[ a_2 \]

\[ a_3 \]

\[ a_5 \]

\[ a_f \]
\( \mathbf{D}^\# = \langle D^\#, \sqsubseteq, \sqcap, \ldots \rangle \),

\( T_{\langle l_i, l_j \rangle} : D^\# \rightarrow D^\# \) a monotonic transfer function (for any edge \( \langle l_i, l_j \rangle \))

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

\[
\begin{align*}
T_{\langle l_1, l_2 \rangle} (a_1) & \sqsubseteq a_2 \\
T_{\langle l_2, l_5 \rangle} (a_2) & \sqsubseteq a_5 \\
T_{\langle l_1, l_f \rangle} (a_1) & \sqsubseteq a_f \\
\ldots 
\end{align*}
\]

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

\[ \alpha(T_e d) \sqsubseteq T_e^\# (\alpha d) \]
A partial annotation map is a partial mapping $S : \mathcal{P} \rightarrow \mathcal{A}$.

- partial annotations generalize stackmaps

May be extended to $\hat{S} : \mathcal{P} \rightarrow \mathcal{A}$

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

- provided the domain of $S$ is sufficiently large

However checking $\sqsubseteq$ may be...

- Expensive
- Undecidable
\[\langle \{a_1 \ldots a_n\}, c \rangle\] is a certified solution if for any edge \(\langle i, j \rangle\)
\[c(i, j) \in C(\vdash T_{\langle i, j \rangle}(a_i) \sqsubseteq a_j)\]

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

If \(\{a_1 \ldots a_n\}\) is a solution of \((D^\#, T^\#)\), and cons s.t. for any edge \(\langle i, j \rangle\)
\[\text{cons}_{\langle i, j \rangle} \in C(\vdash T_{\langle i, j \rangle}(\gamma(a)) \sqsubseteq \gamma(T_{\langle i, j \rangle}^\#(a)))\]

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

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

**Verified abstraction carrying code**

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

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

- Generic lattice library
- General lemmas about well-founded orders
Example of certified analyzer: memory consumption

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

**Transfer rules**

\[
\begin{align*}
P[i] = \text{new, newarray} & \\
i \vdash n \Rightarrow n + 1
\end{align*}
\]

\[
\begin{align*}
P[i] \neq \text{new, newarray} & \\
i \vdash n \Rightarrow n
\end{align*}
\]
Bounded programs

Typing rule for source level programs:

\[
\frac{c : 0 \quad \text{while } b \text{ do } c : 0}{c : 0}
\]

One can enforce a similar constraint for bytecode using widening:

- A program is bounded iff for every \( i \) s.t. \( P[i] = \text{new, newarray} \), \( i \) is not in a loop, i.e. \( i \not\rightarrow^+ i \)
- Assume \( P \) is safe. Then \( P \) is bounded iff there exists \( n \) s.t. \( \vdash P : n \).
"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
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982

\[ \forall s_1, s_2, s_1 \sim_L s_2 \land P, s_1 \downarrow s_1' \land P, s_2 \downarrow s_2' \implies s_1' \sim_L s_2' \]

High = confidential \hspace{1cm} Low = public
A program is an array of instructions:

\[
\text{instr} ::= \begin{array}{l}
\text{prim op} & \text{primitive operation} \\
\mid \text{push } v & \text{push value on top of stack} \\
\mid \text{load } x & \text{load value of } x \text{ on stack} \\
\mid \text{store } x & \text{store top of stack in } x \\
\mid \text{if } j & \text{conditional jump} \\
\mid \text{goto } j & \text{unconditional jump} \\
\mid \text{return} & \text{return}
\end{array}
\]

where:

- \( j \in \mathcal{P} \) is a program point
- \( v \in \mathcal{V} \) is a value
- \( x \in \mathcal{X} \) is a variable
States are of the form $\langle i, \rho, s \rangle$ where:
- $i : \mathcal{P}$ is the program counter
- $\rho : \mathcal{X} \rightarrow \mathcal{V}$ maps variables to values
- $s : \mathcal{V}^*$ is the operand stack

Operational semantics is given by rules are of the form

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

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

\[
\begin{align*}
P[i] &= \text{prim } op \quad n_1 \text{ op } n_2 = n \\
\langle\langle i, \rho, n_1 :: n_2 :: s \rangle\rangle &\leadsto \langle\langle i + 1, \rho, n :: s \rangle\rangle \\
P[i] &= \text{load } x \\
\langle\langle i, \rho, s \rangle\rangle &\leadsto \langle\langle i + 1, \rho, \rho(x) :: s \rangle\rangle \\
P[i] &= \text{if } j \\
\langle\langle i, \rho, \text{false} :: s \rangle\rangle &\leadsto \langle\langle j, \rho, s \rangle\rangle \\
P[i] &= \text{goto } j \\
\langle\langle i, \rho, s \rangle\rangle &\leadsto \langle\langle j, \rho, s \rangle\rangle \\
P[i] &= \text{push } n \\
\langle\langle i, \rho, s \rangle\rangle &\leadsto \langle\langle i + 1, \rho, n :: s \rangle\rangle \\
P[i] &= \text{store } x \\
\langle\langle i, \rho, v :: s \rangle\rangle &\leadsto \langle\langle i + 1, \rho(x := v), s \rangle\rangle \\
P[i] &= \text{if } j \\
\langle\langle i, \rho, \text{true} :: s \rangle\rangle &\leadsto \langle\langle i + 1, \rho, s \rangle\rangle \\
P[i] &= \text{return} \\
\langle\langle i, \rho, v :: s \rangle\rangle &\leadsto \langle\langle \rho, v \rangle\rangle
\end{align*}
\]
Examples of insecure programs

<table>
<thead>
<tr>
<th>Direct flow</th>
<th>Indirect flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>load (y_H)</td>
<td>load (y_H)</td>
</tr>
<tr>
<td>store (x_L)</td>
<td>if 5</td>
</tr>
<tr>
<td>return</td>
<td>push 0</td>
</tr>
<tr>
<td></td>
<td>store (x_L)</td>
</tr>
<tr>
<td></td>
<td>return</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flow via return</th>
<th>Flow via operand stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>load (y_H)</td>
<td>push 0</td>
</tr>
<tr>
<td>if 5</td>
<td>push 1</td>
</tr>
<tr>
<td>push 1</td>
<td>load (y_H)</td>
</tr>
<tr>
<td>return</td>
<td>if 6</td>
</tr>
<tr>
<td>push 0</td>
<td>swap</td>
</tr>
<tr>
<td>return</td>
<td>store (x_L)</td>
</tr>
<tr>
<td>return</td>
<td>return 0</td>
</tr>
</tbody>
</table>
A lattice of security levels $S = \{H, L\}$ with $L \leq H$

Each program is given a security signature: $\Gamma : X \rightarrow S$ and $k_{ret}$.

$\Gamma$ determines an equivalence relation $\sim_L$ on memories: $\rho \sim_L \rho'$ iff

$$\forall x \in X. \Gamma(x) \leq L \Rightarrow \rho(x) = \rho'(x)$$

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

$$P, \mu \downarrow \nu, v$$

$$P, \mu' \downarrow \nu', v'$$

$$\mu \sim_L \mu'$$

$$\Rightarrow \nu \sim_L \nu' \land (k_{ret} \leq L \Rightarrow v = v')$$
Type system

- Transfer rules of the form

\[
P[i] = \text{ins constraints} \quad i \vdash st \Rightarrow st' \quad P[i] = \text{ins constraints} \quad i \vdash st' \Rightarrow
\]

where \( st, st' \in S^* \).

- Types assign stack of security levels to program points

\[
S : P \rightarrow S^*
\]

\[
S \vdash P \text{ iff } S_1 = \epsilon \text{ and for all } i, j \in P
\]

- \( i \mapsto j \Rightarrow \exists st'. i \vdash S_i \Rightarrow st' \land st' \leq S_j; \)

\( i \mapsto \Rightarrow i \vdash S_i \Rightarrow \)

The transfer rules and typability relation are implicitly parametrized by a signature \( \Gamma, k_{\text{ret}} \) 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 $\text{region}(i)$
- $b$ does not necessary belong to $\text{region}(i)$
In a typical type system for a structured language:

\[
\begin{align*}
\vdash & \text{exp} : k \\
[k_1] & \vdash c_1 \\
[k_2] & \vdash c_2 \\
k & \leq k_1 \\
k & \leq k_2
\end{align*}
\]

\[
[k] \vdash \text{if exp then } c_1 \text{ else } c_2
\]

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 \( j \in \text{region}(i) \)

\[
P[i] = \text{if } i' \quad \forall j \in \text{region}(i), \; k \leq se(j)
\]

\[
i \vdash k :: st \Rightarrow \cdots
\]
CDR soundness is ensured by local conditions (instead of path properties) using \( \text{region} \in \mathcal{P} \to \wp(\mathcal{P}) \) and \( \text{jun} \in \mathcal{P} \to \mathcal{P} \).

**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), \( k \in \text{region}(i) \) or \( k = \text{jun}(i) \);

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

**SOAP3:** for all program points \( i, j \), if \( j \in \text{region}(i) \) and \( j \mapsto \) then \( \text{jun}(i) \) is undefined.
CDR soundness is ensured by local conditions (instead of path properties) using \( \text{region} \in \mathcal{P} \rightarrow 2^{\mathcal{P}} \) and \( \text{jun} \in \mathcal{P} \rightarrow \mathcal{P} \).

**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), \( k \in \text{region}(i) \) or \( k = \text{jun}(i) \);

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

**SOAP3:** for all program points \( i, j \), if \( j \in \text{region}(i) \) and \( j \mapsto \) then \( \text{jun}(i) \) is undefined.
CDR soundness is ensured by local conditions (instead of path properties) using \( \text{region} \in \mathcal{P} \rightarrow \wp(\mathcal{P}) \) and \( \text{jun} \in \mathcal{P} \rightarrow \mathcal{P} \).

**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), \( k \in \text{region}(i) \) or \( k = \text{jun}(i) \);

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

**SOAP3:** for all program points \( i, j \), if \( j \in \text{region}(i) \) and \( j \mapsto \) then \( \text{jun}(i) \) is undefined.
CDR soundness is ensured by local conditions (instead of path properties) using $\text{region} \in \mathcal{P} \rightarrow \wp(\mathcal{P})$ and $\text{jun} \in \mathcal{P} \rightarrow \mathcal{P}$.

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

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

**SOAP3:** for all program points $i, j$, if $j \in \text{region}(i)$ and $j \mapsto$ then $\text{jun}(i)$ is undefined.
CDR soundness is ensured by local conditions (instead of path properties) using $\text{region} \in \mathcal{P} \rightarrow \wp(\mathcal{P})$ and $\text{jun} \in \mathcal{P} \rightarrow \mathcal{P}$.

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

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

**SOAP3:** for all program points $i, j$, if $j \in \text{region}(i)$ and $j \not\mapsto$ then $\text{jun}(i)$ is undefined.