Semantics of Programming Languages

Operational Semantics

Prof. Peter Müller

Software Component Technology

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

2. Operational Semantics

Static Safety

- Realistic programming languages support many different kinds of values
- Most operations are only defined for certain kinds of values
- **Type systems** check statically that operations are only applied to values for which they are defined

```
procedure foo(p,q; res)
begin
  res := p + q
end
```

```
var res := 0 in
  foo(5,2;res)
end
```

```
var res := 0 in
  foo(5,(5>2);res)
end
```

Static Safety: Example

```
procedure foo(p: int,q: int; res: int)
begin
  res := p + q
end
```

```
var res: int := 0 in
  foo(5,2;res)
end
```

```
var res: int := 0 in
  foo(5,(5>2);res)
end
```

Type System

Definition: A type system is a syntactic method for proving absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

[B.C. Pierce, 2002]

- *Syntactic:* Rules can be checked by a compiler
- *Phrases:* Expressions, methods, etc. of a program
- *Kinds of values:* Types

Types

Definition: A type is a set of values sharing some properties. A value x has type T if x is an element of T

- Properties: Available operations, etc.
- Remarks:
  - Most languages provide primitve types (boolean, int, char, etc.) and user-defined types (records, classes, etc.)
  - The subtype relation on types corresponds to the subset relation on sets of values

Type Ckecking

- **Each expression** of a program **has a type**
- **Types** of variables are **declared explicitly**
- Types of expressions are derived from the types of their constituents
- **Type rules** check whether a phrase is **correctly typed**

```
"A String"
5+7
```

```
a: bool
procedure
 equals(p,q:String;b:bool)
```

```
a and true
equals("hello","test";a)
```

Declaration Environments

- The set of all types of a program is Type
- A **declaration environment** associates a type to each variable
- Declaration environments are represented as finite functions Var $\longrightarrow$ Type
- Example: $\Gamma = \{x_1 \mapsto T_1, x_2 \mapsto T_2, \ldots, x_n \mapsto T_n\}$
- For languages with procedures, the declaration environment would also contain the signatures of all procedures

Type Judgments

- **Type judgment for expressions**

$$\Gamma \vdash e :: T$$

Meaning: expression e is well-typed in environment Γ and has type T

- **Type judgment for statements**

$$\Gamma \vdash s$$

Meaning: statement s is well-typed in environment Γ

Type Rules

- The valid type judgments are described by a set of **type axioms and rules**
- Examples:

A local variable access x is correctly typed and has the declared type of x

$$\Gamma \vdash x :: \Gamma(x)$$

An assignment $x := e$ is correctly typed if

- e is correctly typed
- e and x have the same type

$$\frac{\Gamma \vdash e :: T, \quad \Gamma(x) = T}{\Gamma \vdash x := e}$$

Static Type Safety

Definition: A programming language is called type-safe if its design prevents type errors

- Type-safe languages guarantee the following **type invariant**:

In every execution state, the type of the value held by variable x is the declared type of x

- Type safety guarantees the absence of certain runtime errors
- IMP's syntax guarantees type safety, even without a type system

Type Invariant

- We can introduce a function that yields the type of a value: $typeof$: Val $\longrightarrow$ Type
  - Example: $typeof(5) = \texttt{int}$

- Type invariant: The following property holds in each execution state σ of a program with declaration environment Γ

$$\forall x : typeof(\sigma(x)) = \Gamma(x)$$

  - σ and Γ map the same local variables to values and types, resp.

Discussion

- Advantages of static type checking
  - Robustness: **Elimination of type errors**
  - Readability: Types are **excellent documentation**
  - Efficiency: Type information allows **optimizations**

- Limitations: Static type checking is only an **approximation of the behavior at runtime**
  - Some programs are rejected by the type checker although they would never cause a runtime error

```
a := 5;    a := a + 5;
a := true; a := a and a
```

2. Operational Semantics

Secrecy

- Programs hold confidential and non-confidential information
- Attackers
  - should not be able to get (partial) information about confidential information
  - know the code of the program
  - know the initial and final values of non-confidential information
- Example: Attackers can read variable `l`, but not h

```
private int h; // confidential
public  int l; // non-confidential
```

Information Flow

- Access Control restricts the release of information, but not its propagation

```
private int h; // confidential
public  int l; // non-confidential
```

- Explicit information flow

```
l := h
```

- Implicit information flow

```
if h > 0 then  l := 1
         else  l := -1
end
```

Noninterference

- To keep data confidential, the following **noninterference policy** should be enforced

An attacker cannot observe any difference between two executions that differ only in their confidential input

```
l := h
```

```
if h > 0 then  l := 1
         else  l := -1
end
```

Formalization: Preliminaries

- We group all program variables into two sets H and L
  - Values of variables in H (**high variables**) are confidential
  - Values of variables in L (**low variables**) are non-confidential

- The equivalence relation $\equiv_L$ describes that two states have the same values for all low variables:

$$\sigma \equiv_L \sigma' \Leftrightarrow \forall x \in L : \sigma(x) = \sigma'(x)$$

Formalization: Noninterference

- The relation $\approx_L$ expresses the **observational power** of an attacker:

$$\sigma \approx_L \sigma' \Leftrightarrow (\sigma, \sigma' \in \text{State} \Rightarrow \sigma \equiv_L \sigma')$$

  - Attackers do not observe termination

- A statement s satisfies the noninterference property iff:

$$\forall \sigma_1, \sigma_2 : \sigma_1 \equiv_L \sigma_2 \Rightarrow \mathcal{S}_{NS}[\![s]\!]\sigma_1 \approx_L \mathcal{S}_{NS}[\![s]\!]\sigma_2$$

  - If two input states share the same low values, then the behaviors of the statement executed on these states are indistinguishable by the attacker

Noninterference: Examples

$$\forall \sigma_1, \sigma_2 : \sigma_1 \equiv_L \sigma_2 \Rightarrow \mathcal{S}_{NS}[\![s]\!]\sigma_1 \approx_L \mathcal{S}_{NS}[\![s]\!]\sigma_2$$

- Assume l $\in L$ and h $\in H$

```
h := l + 4
```

```
l := h
```

```
if l = 5 then h := h + 1
         else l := l + 1
end
```

```
if h = 3 then l := 5
         else skip
end
```

- Explanation:
  - Let $\sigma_1 = \{\mathtt{l} \mapsto 0, \mathtt{h} \mapsto 2\}$ and $\sigma_2 = \{\mathtt{l} \mapsto 0, \mathtt{h} \mapsto 3\}$

Summary

- The formal semantics allows us to express security properties such as noninterference
- More fine-grained policies can be achieved by
  - Additional confidentiality levels (not only high and low)
  - Different relations $\approx_L$ for observational power

- The formalization enables one to prove that a statement has the noninterference property
- Interesting question:
  Can we check noninterference syntactically?

2. Operational Semantics

Approach

- Use a **type system** to check noninterference
- Variables have **security type** `high` or `low`
  - Handles explicit information flow

```
l := h
```

- Statements have **security context** `high` or `low`
  - If the control flow depends on high values, the security context is high
  - In high contexts, it is not allowed to assign to low variables
  - Handles implicit information flow

```
if h = 3 then l := 5  else skip end
```

Type Judgments

- **Type judgment for expressions**

$$\Gamma \vdash e :: T$$

- **Type judgment for statements**

$$\Gamma, \Delta \vdash s$$

- Types: Type $= \{\texttt{high}, \texttt{low}\}$
- Declaration environments: Γ : Var $\longrightarrow$ Type
- Security contexts: $\Delta \in \{\texttt{high}, \texttt{low}\}$

Type Rules: Expressions

- It is always safe to type expressions as `high`

$$\Gamma \vdash b :: \texttt{high} \qquad \Gamma \vdash e :: \texttt{high}$$

- An expression can have type `low` only if it does not contain high variables

$$\frac{\forall x \in FV(b) : \Gamma(x) \neq \texttt{high}}{\Gamma \vdash b :: \texttt{low}}$$

$$\frac{\forall x \in FV(e) : \Gamma(x) \neq \texttt{high}}{\Gamma \vdash e :: \texttt{low}}$$

Type Rules: `skip` and Assignment

- `skip` can be typed in any security context

$$\Gamma, \Delta \vdash \texttt{skip}$$

- Assignments to high variables are always safe

$$\frac{\Gamma(x) = \texttt{high}}{\Gamma, \Delta \vdash x\texttt{:=}e}$$

- Assignments to low variables are only possible if the security context is `low`

$$\frac{\Gamma \vdash e :: \texttt{low}, \quad \Gamma(x) = \texttt{low}}{\Gamma, \texttt{low} \vdash x\texttt{:=}e}$$

Type Rules: Conditional and Loop

- If the condition of an if-statement depends on high values, then the security context of the statement has to be `high`

$$\frac{\Gamma \vdash b :: \Delta, \quad \Gamma, \Delta \vdash s_1, \quad \Gamma, \Delta \vdash s_2}{\Gamma, \Delta \vdash \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}}$$

- Loops are typed analogously

$$\frac{\Gamma \vdash b :: \Delta, \quad \Gamma, \Delta \vdash s}{\Gamma, \Delta \vdash \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}}$$

Type Rules: Seq. Composition

- For sequential composition, both statements must have the same security context

$$\frac{\Gamma, \Delta \vdash s_1, \quad \Gamma, \Delta \vdash s_2}{\Gamma, \Delta \vdash s_1 \texttt{;} s_2}$$

- If a statement can be typed in security context `high` then it can also be typed in security context `low` (subsumption)

$$\frac{\Gamma, \texttt{high} \vdash s}{\Gamma, \texttt{low} \vdash s}$$

Subsumption: Example

- Assume $\Gamma = \{\texttt{l} \mapsto \texttt{low}, \texttt{h} \mapsto \texttt{high}\}$
- The statement

```
if l = 0 then l := l + 1 end
```

  can only be typed in security context `low`
- Without using subsumption, the statement

```
if h = 0 then h := h + 1 end
```

  can only be typed in security context `high`
- To be able to type the sequential composition the subsumption rule has to be used

```
if l = 0 then l := l + 1 end;
if h = 0 then h := h + 1 end
```

Type Rules: Examples

- Assume $\Gamma = \{\texttt{l} \mapsto \texttt{low}, \texttt{h} \mapsto \texttt{high}\}$

```
h := l + 4
```

Typeable in high and low

```
l := h
```

Not typeable

```
if l = 5 then h := h + 1
         else l := l + 1
end
```

Typeable in low

```
if h = 3 then l := 5
         else skip
end
```

Not typeable

Type Safety

- The type system does not have the usual type invariant
  - Values are not per se high or low values
  - Classical type safety is not applicable

- The type system guarantees the noninterference porperty as type invariant:

$$\forall s, \sigma_1, \sigma_2 : \sigma_1 \equiv_L \sigma_2 \Rightarrow \mathcal{S}_{NS}[\![s]\!]\sigma_1 \approx_L \mathcal{S}_{NS}[\![s]\!]\sigma_2$$

where $\equiv_L$ is defined as follows

$$\sigma \equiv_L \sigma' \Leftrightarrow (\forall x : \Gamma_s(x) = \texttt{low} \Rightarrow \sigma(x) = \sigma'(x))$$

Lemmas: Expressions

- The proof of type safety uses the following lemmas

- The values of expressions of type `low` do not depend on values of high variables

Lemma 2.4.1

$$\Gamma \vdash e :: \texttt{low} \wedge \sigma_1 \equiv_L \sigma_2 \Rightarrow \mathcal{A}[\![e]\!]\sigma_1 = \mathcal{A}[\![e]\!]\sigma_2$$
$$\Gamma \vdash b :: \texttt{low} \wedge \sigma_1 \equiv_L \sigma_2 \Rightarrow \mathcal{B}[\![b]\!]\sigma_1 = \mathcal{B}[\![b]\!]\sigma_2$$

  - The proofs run by structural induction on e and b

Lemmas (cont'd)

- $\equiv_L$ is reflexive, symmetric, and transitive

Lemma 2.4.2
$\equiv_L$ is an equivalence relation

  - The lemma follows directly from equality "=" being an equivalence relation

- Statements that can be typed in security context `high` do not modify the values of low variables

Lemma 2.4.3
$$\Gamma, \texttt{high} \vdash s \wedge \langle s, \sigma \rangle \rightarrow \sigma' \Rightarrow \sigma \equiv_L \sigma'$$

  - See Exercise 20 for the proof

Proof: Overview

- We have to prove the type invariant

$$\forall s, \sigma_1, \sigma_2 : \sigma_1 \equiv_L \sigma_2 \Rightarrow \mathcal{S}_{NS}[\![s]\!]\sigma_1 \approx_L \mathcal{S}_{NS}[\![s]\!]\sigma_2$$

- We prove the following equivalent property

$$\begin{aligned}&\forall s, \sigma_1, \sigma_2, \sigma_1', \sigma_2' : \sigma_1 \equiv_L \sigma_2 \wedge \\ &\langle s, \sigma_1 \rangle \rightarrow \sigma_1' \wedge \langle s, \sigma_2 \rangle \rightarrow \sigma_2' \\ &\Rightarrow \sigma_1' \equiv_L \sigma_2'\end{aligned}$$

- The type safety proof runs by induction on the shape of the derivation tree for $\langle s, \sigma_1 \rangle \rightarrow \sigma_1'$

Induction Base: `skip`

- Case skip-axiom: We know
  - $s = \texttt{skip}$
  - $\langle \texttt{skip}, \sigma_1 \rangle \rightarrow \sigma_1'$ implies $\sigma_1' = \sigma_1$
  - $\langle \texttt{skip}, \sigma_2 \rangle \rightarrow \sigma_2'$ implies $\sigma_2' = \sigma_2$
- These equalities trivially imply $\sigma_1 \equiv_L \sigma_2 \Rightarrow \sigma_1' \equiv_L \sigma_2'$

Induction Base: Assignment

- Case assign-axiom: The derivation tree is the axiom instance $\langle x := e, \sigma_1 \rangle \rightarrow \sigma_1'$ and we know
  - $s = x := e$
  - $\sigma_1' = \sigma_1[x \mapsto \mathcal{A}[\![e]\!]\sigma_1]$ and $\sigma_2' = \sigma_2[x \mapsto \mathcal{A}[\![e]\!]\sigma_2]$
- Case 1: $\Gamma(x) = \texttt{high}$
  - $\sigma_1' = \sigma_1[x \mapsto \mathcal{A}[\![e]\!]\sigma_1] \Rightarrow \sigma_1' \equiv_L \sigma_1$
  - $\sigma_2' = \sigma_2[x \mapsto \mathcal{A}[\![e]\!]\sigma_2] \Rightarrow \sigma_2' \equiv_L \sigma_2$
  - Lemma 2.4.2 and $\sigma_1' \equiv_L \sigma_1 \equiv_L \sigma_2 \equiv_L \sigma_2'$ imply $\sigma_1' \equiv_L \sigma_2'$
- Case 2: $\Gamma(x) = \texttt{low}$
  - From the type rule we get $\Gamma \vdash e :: \texttt{low}$
  - By Lemma 2.4.1, we get $\mathcal{A}[\![e]\!]\sigma_1 = \mathcal{A}[\![e]\!]\sigma_2$
  - $\sigma_1 \equiv_L \sigma_2 \Rightarrow \sigma_1[x \mapsto \mathcal{A}[\![e]\!]\sigma_1] \equiv_L \sigma_2[x \mapsto \mathcal{A}[\![e]\!]\sigma_2] \Rightarrow \sigma_1' \equiv_L \sigma_2'$

Induction Step: Seq. Composition

- Case sequence-rule: The root of the derivation tree is $\langle s_1 ; s_2, \sigma_1 \rangle \rightarrow \sigma_1'$.
  - There are derivation trees for $\langle s_1, \sigma_1 \rangle \rightarrow \sigma_1''$ and $\langle s_2, \sigma_1'' \rangle \rightarrow \sigma_1'$ for some state σ_1''
  - There are derivation trees for $\langle s_1, \sigma_2 \rangle \rightarrow \sigma_2''$ and $\langle s_2, \sigma_2'' \rangle \rightarrow \sigma_2'$ for some state σ_2''
  - By applying the induction hypothesis to $\langle s_1, \sigma_1 \rangle \rightarrow \sigma_1''$ and $\langle s_1, \sigma_2 \rangle \rightarrow \sigma_2''$ we get $\sigma_1'' \equiv_L \sigma_2''$
  - By applying the induction hypothesis to $\langle s_2, \sigma_1'' \rangle \rightarrow \sigma_1'$ and $\langle s_2, \sigma_2'' \rangle \rightarrow \sigma_2'$ we get $\sigma_1' \equiv_L \sigma_2'$

Induction Step: `if`

- Case if-rule: The root of the derivation tree is $\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma_1 \rangle \rightarrow \sigma_1'$
- Case 1: $\Gamma \vdash b :: \texttt{high}$
  - By the type rules, we get that s has security context `high`
  - By Lemma 2.4.3, we get $\sigma_1 \equiv_L \sigma_1'$ and $\sigma_2 \equiv_L \sigma_2'$
  - By Lemma 2.4.2, we get $\sigma_1' \equiv_L \sigma_2'$
- Case 2: $\Gamma \vdash b :: \texttt{low}$
  - By Lemma 2.4.1, we get $\mathcal{B}[\![b]\!]\sigma_1 = \mathcal{B}[\![b]\!]\sigma_2$
  - If $\mathcal{B}[\![b]\!]\sigma_1 = tt$, we have $\langle s_1, \sigma_1 \rangle \rightarrow \sigma_1'$ and $\langle s_1, \sigma_2 \rangle \rightarrow \sigma_2'$
  - By applying the induction hypothesis, we get $\sigma_1' \equiv_L \sigma_2'$
  - The case for $\mathcal{B}[\![b]\!]\sigma_1 = ff$ is analogous

Induction Step: `while`

- Case while-rule: The root of the derivation tree is $\langle \texttt{while}\ b\ \texttt{do}\ s'\ \texttt{end}, \sigma_1 \rangle \rightarrow \sigma_1'$
- Case 1: $\Gamma \vdash b :: \texttt{high}$
  - By the type rules, we get that s has security context `high`
  - By Lemma 2.4.3, we get $\sigma_1 \equiv_L \sigma_1'$ and $\sigma_2 \equiv_L \sigma_2'$
  - By Lemma 2.4.2, we get $\sigma_1' \equiv_L \sigma_2'$ and

Induction Step: `while`

- Case 2: $\Gamma \vdash b :: \texttt{low}$
  - By Lemma 2.4.1, we get $\mathcal{B}[\![b]\!]\sigma_1 = \mathcal{B}[\![b]\!]\sigma_2$
  - If $\mathcal{B}[\![b]\!]\sigma_1 = tt$, we have
    $\langle s', \sigma_1 \rangle \rightarrow \sigma_1''$ and $\langle \texttt{while}\ b\ \texttt{do}\ s'\ \texttt{end}, \sigma_1'' \rangle \rightarrow \sigma_1'$
    as well as
    $\langle s', \sigma_2 \rangle \rightarrow \sigma_2''$ and $\langle \texttt{while}\ b\ \texttt{do}\ s'\ \texttt{end}, \sigma_2'' \rangle \rightarrow \sigma_2'$
  - By applying the induction hypothesis to $\langle s', \sigma_1 \rangle \rightarrow \sigma_1''$ and $\langle s', \sigma_2 \rangle \rightarrow \sigma_2''$, we get $\sigma_1'' \equiv_L \sigma_2''$
  - By applying the induction hypothesis to
    $\langle \texttt{while}\ b\ \texttt{do}\ s'\ \texttt{end}, \sigma_1'' \rangle \rightarrow \sigma_1'$ and
    $\langle \texttt{while}\ b\ \texttt{do}\ s'\ \texttt{end}, \sigma_2'' \rangle \rightarrow \sigma_2'$, we get $\sigma_1' \equiv_L \sigma_2'$
  - The case for $\mathcal{B}[\![b]\!]\sigma_1 = ff$ is trivial

Discussion

- The type system can be used to check noninterference statically
- Like all type systems, it is a **static approximation** of the semantics
  - It rejects statements that are safe
- Example

```
l := h; l := l - h
```

  - The statement is not typeable
  - However, it is secure ($\sigma'(\mathtt{l}) = 0$ for all inputs h)

References

- D. Volpano, C. Irvine, G. Smith: *A sound type system for secure flow analysis*. Journal of Computer Security, 4(2,3):167–187, 1996
- A. Sabelfeld and A. C. Myers: *Language-Based Information-Flow*. Security IEEE Journal on Selected Areas in Communications, 21(1):5–19, 2003
- D. Denning: *A Lattice Model of Secure Information Flow*. Communications of the ACM 20(7):236–242, 1976

- The papers are available on the course web site

Summary: Operational Semantics

- Operational semantics describe how the effects of a computation are achieved
  - Close to the intuition about languages
  - Can be executable
  - Simple mathematical background

- Main forms
  - Natural semantics, with predominant proof principle “induction on the shape of derivation trees”
  - Structural operational semantics, with predominant proof principle “induction on the length of derivation sequences”
  - Abstract state machines will be presented by Robert Stärk