Type Systems

DD2460 Software Safety and Security: Part III, lecture 2

Gurvan Le Guernic

DD2460 (III, L2)
February 17th, 2012

Outline

Type systems are used to *verify some properties* that any execution of the analyzed program should have.

Type Systems

What is a type?

A type is the *describing name* of a set of entities.

Examples:

- all self-propelled 4 wheels light vehicles belongs to the type “car”
- all elements in $\mathbb{N}$ belongs to the type “natural number”
- all elements in $\mathbb{Z}$ belongs to the type “integer”
- in Java:
  - all elements in $\{-128, -127, \ldots, 126, 127\}$ belongs to the type `byte`
  - all elements in $\{-(2^{31}), \ldots, (2^{31}-1)\}$ belongs to the type `int`
  - all elements in $\{$`true`, `false`$\}$ belongs to the type `boolean`

An element can be of different types

- 0 belongs to the types `byte` and `int`

Statically vs dynamically typed languages

Variable types are used to give meaning to the sequence of bits that the variable will contain during execution

Statically typed languages (ex. Java) use types at compile-time, mainly to rule out programs that could end-up in a non-meaningful state

- x := “Hello”; y := 3 / x
- x := “Hello”; **if** (x) {...} **else** {...}

Dynamically typed languages (ex. Python) use types at run-time, mainly to direct the run-time semantics

- x + y
  - addition if x and y are numbers
  - concatenation if at least one of x and y is a string

What is a type system? (1)

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

B. C. Pierce. "Types and Programming Languages"

A system that relates (syntactically, usually at compile-time) sub-parts (variables/statements) of a program to a type that reflects the potential run-time values/behaviors of the associated sub-part.

- Variable x is of type `int` implies that, at run-time, x will only contain values in $[-(2^{31}), (2^{31}-1)]$
- If statement types reflect only "good" behaviors, then a program that types can not go "wrong".

What is a type system? (2)

A set of rules which, applied on a given program, are true if and only if a type (that reflect its potential run-time values/behaviors) can be given to every relevant sub-parts of the program.

A static analysis technique which is:

- automatic (sometimes using type inference)
- scalable (usually linear in the size of the program)
- easier but less expressive (?) than many other static analysis techniques

While language

$$\begin{array}{rcl} e & ::= & \text{true} \mid \text{false} \mid n \mid x \mid e \odot e \mid e \oplus e \mid e \otimes e \\ S & ::= & \text{skip} \mid x := e \mid S; S \\ & \mid & \textbf{if } (e)\,\{S\}\,\textbf{else}\,\{S\} \\ & \mid & \textbf{while } (e)\,\{S\} \end{array}$$

where:

- $\odot$ is a boolean binary operator (and, or, xor, . . .) (not x = x xor false)
- $\oplus$ is a “number” binary operator returning a boolean (==, <, . . .)
- $\otimes$ is a “number” binary operator (+, -, ×, . . .) (maybe not /)

Small step semantics (1)

Let σ be a memory and $\sigma(e)$ the value of the “well-formed” expression e when evaluated with that memory

Execution of S with memory σ yielding new statement S' and memory σ':

$$(S,\ \sigma) \rightarrow (S',\ \sigma') \qquad \text{or} \qquad (S,\ \sigma) \rightarrow \sigma'$$

$$(\text{skip},\ \sigma) \rightarrow \sigma \qquad\qquad (x := e,\ \sigma) \rightarrow \sigma[x \mapsto \sigma(e)]$$

$$\frac{(S_1,\ \sigma) \rightarrow \sigma'}{(S_1 ; S_2,\ \sigma) \rightarrow (S_2,\ \sigma')} \qquad\qquad \frac{(S_1,\ \sigma) \rightarrow (S_1',\ \sigma')}{(S_1 ; S_2,\ \sigma) \rightarrow (S_1' ; S_2,\ \sigma')}$$

Small step semantics (2)

$$\frac{\sigma(e) = \text{true}}{(\,\textbf{if}\ \ (\,e\,)\,\{\,S_1\,\}\ \textbf{else}\ \{\,S_2\,\},\ \sigma\,) \rightarrow (\,S_1,\ \sigma\,)}$$

$$\frac{\sigma(e) = \text{false}}{(\,\textbf{if}\ \ (\,e\,)\,\{\,S_1\,\}\ \textbf{else}\ \{\,S_2\,\},\ \sigma\,) \rightarrow (\,S_1,\ \sigma\,)}$$

$$\frac{\sigma(e) = \text{true}}{(\textbf{while}\ (\,e\,)\ \{\,S\},\ \sigma\,) \rightarrow (\,S; \textbf{while}\ (\,e\,)\ \{\,S\},\ \sigma\,)}$$

$$\frac{\sigma(e) = \text{false}}{(\textbf{while}\ (\,e\,)\ \{\,S\},\ \sigma\,) \rightarrow \sigma}$$

Run time error

Evaluation of statement *S* is “stuck” if and only if *S* contains an “ill-formed” expression or statement

Ill-formed expression or statement:

- expression or statement that can not be evaluated
- examples:
  - $\text{true} \oplus 1.5$
  - **if** $(1.5 \otimes 2)$ $\{\ldots\}$ **else** $\{\ldots\}$

Type systems are used (among other things) to rule out, at compile-time, programs that can get “stuck” (evaluate to a statement containing an ill-formed sub component).

Typing rules (1)

Let Γ be a typing environment (i.e. a mapping from variables to types) and $\Gamma(x)$ the type of variable x

Typing e with type t (`bool` or `float`) and S with type `cmd` in the environment Γ: $\Gamma \vdash e : t$ and $\Gamma \vdash S : \texttt{cmd}$

$$\Gamma \vdash n : \texttt{float} \qquad \frac{t = \Gamma(x)}{\Gamma \vdash x : t} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \odot e_2 : \texttt{bool}}$$

$$\Gamma \vdash \text{true} : \texttt{bool} \qquad \frac{\Gamma \vdash e_1 : \texttt{float} \qquad \Gamma \vdash e_2 : \texttt{float}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{bool}}$$

$$\Gamma \vdash \text{false} : \texttt{bool} \qquad \frac{\Gamma \vdash e_1 : \texttt{float} \qquad \Gamma \vdash e_2 : \texttt{float}}{\Gamma \vdash e_1 \otimes e_2 : \texttt{float}}$$

Typing rules (2)

$$\Gamma \vdash \mathsf{skip} : \mathtt{cmd} \qquad \frac{\Gamma \vdash S_1 : \mathtt{cmd} \qquad \Gamma \vdash S_2 : \mathtt{cmd}}{\Gamma \vdash S_1 ; S_2 : \mathtt{cmd}}$$

$$\frac{\Gamma \vdash e : \mathtt{bool} \qquad \Gamma \vdash S_1 : \mathtt{cmd} \qquad \Gamma \vdash S_2 : \mathtt{cmd}}{\Gamma \vdash \textbf{if}\ (e)\ \{S_1\}\ \textbf{else}\ \{S_2\} : \mathtt{cmd}}$$

$$\frac{\Gamma(x) = t \qquad \Gamma \vdash e : t}{\Gamma \vdash x := e : \mathtt{cmd}} \qquad \frac{\Gamma \vdash e : \mathtt{bool} \qquad \Gamma \vdash S : \mathtt{cmd}}{\Gamma \vdash \textbf{while}\ (e)\ \{S\} : \mathtt{cmd}}$$

Typing examples

$$\dfrac{\dfrac{\dfrac{\texttt{float} = \texttt{float}}{\Gamma \vdash x : \texttt{float}} \qquad \Gamma \vdash 0 : \texttt{float}}{\Gamma \vdash x \oplus 0 : \texttt{bool}} \qquad \Gamma \vdash S : \texttt{cmd}}{\Gamma = [x \mapsto \texttt{float}] \vdash \textbf{while}\ (x \oplus 0)\ \{S\} : \texttt{cmd}}$$

$$\dfrac{? = \texttt{bool} \qquad \dfrac{\dfrac{\texttt{float} = ?}{\Gamma \vdash x : \texttt{float}} \qquad \Gamma \vdash 0 : \texttt{float}}{\Gamma \vdash x \oplus 0 : \texttt{bool}}}{\Gamma = [x \mapsto \quad ? \quad] \vdash x := x \oplus 0 : \texttt{cmd}}$$

Two main theorems

Theorem 1 (Progress)

A typable (well-typed) statement can always take one execution step.

Theorem 2 (Preservation)

A typable (well-typed) statement that take one execution step yields a typable (well-typed) statement (or terminates).

Corollary 3

A typable (well-typed) statement does not get stuck.

Subtyping (1)

What if we also have `int`? $\Gamma \vdash i : \texttt{int}$ $\Gamma \vdash i.i : \texttt{float}$

Solution: subtyping $\texttt{int} \subseteq \texttt{float}$ $\dfrac{\Gamma \vdash e : t \qquad t \subseteq t'}{\Gamma \vdash e : t'}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash 2 : \texttt{int} \qquad \texttt{int} \subseteq \texttt{float}}{\Gamma \vdash 2 : \texttt{float}} \qquad \dfrac{\Gamma \vdash y : \texttt{int} \qquad \texttt{int} \subseteq \texttt{float}}{\Gamma \vdash y : \texttt{float}}}{\Gamma \vdash 2 \otimes y : \texttt{float}}}{[x \mapsto \texttt{float}, y \mapsto \texttt{int}] \vdash x := 2 \otimes y : \texttt{cmd}}$$

To be able to compute an `int`: $\dfrac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t \qquad t \subseteq \texttt{float}}{\Gamma \vdash e_1 \otimes e_2 : t}$

Command types

A type system for termination analysis:

- terminating statement: $\mathtt{T\ \ cmd}$ $\qquad \mathtt{T\ \ cmd} \subseteq \mathtt{cmd}$

In addition to previous rules for sequence, if and while:

$$\frac{\Gamma \vdash S : t \qquad t \subseteq t'}{\Gamma \vdash S : t'} \qquad \Gamma \vdash \mathsf{skip} : \mathtt{T\ \ cmd} \qquad \frac{\Gamma(x) = t \qquad \Gamma \vdash e : t}{\Gamma \vdash x := e : \mathtt{T\ \ cmd}}$$

$$\frac{\Gamma \vdash e : \mathtt{bool} \qquad \Gamma \vdash S_1 : \mathtt{T\ \ cmd} \qquad \Gamma \vdash S_2 : \mathtt{T\ \ cmd}}{\Gamma \vdash \textbf{if}\ (e)\ \{S_1\}\ \textbf{else}\ \{S_2\} : \mathtt{T\ \ cmd}}$$

$$\frac{\Gamma \vdash S_1 : \mathtt{T\ \ cmd} \qquad \Gamma \vdash S_2 : \mathtt{T\ \ cmd}}{\Gamma \vdash S_1 ; S_2 : \mathtt{T\ \ cmd}} \qquad \frac{\Gamma \vdash S : \mathtt{T\ \ cmd}}{\Gamma \vdash \textbf{while}\ (\mathsf{false})\ \{S\} : \mathtt{T\ \ cmd}}$$

Subtyping (2): covariance and contravariance (draft)

Subtyping meaning: $t \subseteq t'$ (t entity = expression/statement of type t)

- $\simeq$ t entity preserves "well-typeness" of `P` when replacing t' entity in `P`
- $\simeq$ t entity can be used where t' entity is "ok"
- examples:
  - if $2.0 \circ 1.5$ does not get stuck, then neither does $2 \circ 1.5$
  - if $\Gamma \vdash P[\textbf{while}\,(e)\,\{S\}] : t\ \texttt{cmd}$ then $\Gamma \vdash P[\text{skip}] : t\ \texttt{cmd}$

Variance: $\Gamma \vdash x := y : \Gamma(x) \leftarrow \Gamma(y)$ $\dfrac{t_2 \subseteq t_2' \qquad t_1' \subseteq t_1}{t_1 \leftarrow t_2 \subseteq t_1' \leftarrow t_2'}$ $(t \leftarrow t' \Rightarrow t' \subseteq t)$

- $t_1 \leftarrow t_2$ is *covariant* on t_2 and *contravariant* on t_1
- $\texttt{TopSecret} \leftarrow \texttt{Public} \subseteq \texttt{Alice} \leftarrow \texttt{Alice}$
- $\texttt{Number} \leftarrow \texttt{int} \subseteq \texttt{float} \leftarrow \texttt{float}$

Security Using Type Systems

Setting up label lattice

```
interface MyLabel {}

interface Top extends MyLabel {}

interface Alice extends Top {}

interface Bob extends Top {}

interface Bottom extends Alice, Bob {}
```

Using generic types to hold labels

```
class MyInteger<L extends MyLabel> {
  private int value;
  public MyInteger(int v) { this.value = v; }

  public <L2 extends L> MyInteger<L2> declassify() {
    return ((MyInteger<L2>) this);
  }

  public <L2 extends L> MyInteger<L> add(MyInteger<L2> i) {
    return new MyInteger<L>( this.value + i.value );
  }

  public static void main(String[] args) {
    MyInteger<Top> topInput; MyInteger<Bottom> bottomInput;
    MyInteger<Alice> aliceInput; MyInteger<Bob> bobInput;

    MyInteger<? extends Top> topOut; MyInteger<? extends Bottom> bottomOut;
    MyInteger<? extends Alice> aliceOut; MyInteger<? extends Bob> bobOut;

    ...
  }
}
```

Using generic types to hold labels

```
aliceInput = new MyInteger<Alice>(2);
bobInput = new MyInteger<Bob>(12);
bottomInput = new MyInteger<Bottom>(6);

// aliceOutput = bobInput;
topOutput = aliceInput;
// bottomOutput = aliceInput;
// bottomOutput = ((MyInteger<Bottom>) aliceInput);
bottomOutput = aliceInput.<Bottom>declassify();

aliceOutput = aliceInput.add(aliceInput);
aliceOutput = aliceInput.add(bottomInput);
topOutput = aliceInput.add(bottomInput);
// bottomOutput = aliceInput.add(aliceInput);
// bottomOutput = aliceInput.add(bottomInput);
// bottomOutput = ((MyInteger<Bottom>) aliceInput).add(bottomInput);
// bottomOutput = bottomInput.add(aliceInput);
// aliceOutput = bottomInput.add(aliceInput);
   // Can't work: L2 must be Alice not to declassify aliceInput
bottomOutput = bottomInput.add(aliceInput.<Bottom>declassify());

MyInteger x; x = aliceInput; x = bobInput; aliceOutput = x;
```

Type System for Noninterference

A type system for noninterference

A type system that can only type noninterfering statements.

$\Gamma \vdash e : l$ only if:

- only variables x such that $\Gamma(x) \rightsquigarrow l$ influence e's value

$\Gamma \vdash S : l\ \texttt{cmd}$ only if:

- S only modifies the value of variables y such that $l \rightsquigarrow \Gamma(y)$

If execution of S is dependent on e's value then:

- x variables influencing e's value also influence y variables modified by S
- however: $\exists l.\ (\Gamma \vdash e : l) \wedge (\Gamma \vdash S : l\ \texttt{cmd}) \Rightarrow \Gamma(x) \rightsquigarrow \Gamma(y)$

$\exists l.\ (\Gamma \vdash S : l\ \texttt{cmd})$ only if:

- input value of x influence output value of y only if $\Gamma(x) \rightsquigarrow \Gamma(y)$

Noninterference type system example (1) (draft)

Let $\bot$ and $\top$ be labels such that: $\forall l.\ \bot \rightsquigarrow l \ \wedge\ l \rightsquigarrow \top$

Subtyping: $$\frac{\Gamma \vdash e : l' \qquad l' \rightsquigarrow l}{\Gamma \vdash e : l} \qquad \frac{\Gamma \vdash S : l' \ \texttt{cmd} \qquad l \rightsquigarrow l'}{\Gamma \vdash S : l \ \texttt{cmd}}$$

$$\Gamma \vdash n : \bot \qquad \Gamma \vdash \text{true} : \bot \qquad \Gamma \vdash \text{false} : \bot$$

$$\frac{l = \Gamma(x)}{\Gamma \vdash x : l} \qquad \frac{\Gamma \vdash e_1 : l \qquad \Gamma \vdash e_2 : l}{\Gamma \vdash e_1 \bigcirc e_2 : l}$$

Noninterference type system example (2) (draft)

Subtyping:
$$\frac{\Gamma \vdash e : l' \qquad l' \rightsquigarrow l}{\Gamma \vdash e : l} \qquad \frac{\Gamma \vdash S : l'\ \mathtt{cmd} \qquad l \rightsquigarrow l'}{\Gamma \vdash S : l\ \mathtt{cmd}}$$

$$\Gamma \vdash \mathsf{skip} : \top\ \mathtt{cmd} \qquad \frac{\Gamma \vdash S_1 : l\ \mathtt{cmd} \qquad \Gamma \vdash S_2 : l\ \mathtt{cmd}}{\Gamma \vdash S_1 ; S_2 : l\ \mathtt{cmd}}$$

$$\frac{\Gamma \vdash e : l \qquad \Gamma \vdash S_1 : l\ \mathtt{cmd} \qquad \Gamma \vdash S_2 : l\ \mathtt{cmd}}{\Gamma \vdash \textbf{if}\ (e)\ \{S_1\}\ \textbf{else}\ \{S_2\} : l\ \mathtt{cmd}}$$

$$\frac{\Gamma(x) = l \qquad \Gamma \vdash e : l}{\Gamma \vdash x := e : l\ \mathtt{cmd}} \qquad \frac{\Gamma \vdash e : l \qquad \Gamma \vdash S : l\ \mathtt{cmd}}{\Gamma \vdash \textbf{while}\ (e)\ \{S\} : l\ \mathtt{cmd}}$$

Wrap-up

5 Most Important Points

- A type (compile-time value) describes a set of run-time values (for expressions) or behaviors (for statements).
- A statement S is well-typed (typable), under a (sound) type system for property P (the behavior corresponding to its type), *only if* P holds for S.
- An expression (or statement) can have multiple types (subtyping).
- Type system $\simeq$ set of rules to verify that a statement is well-typed under a given typing environment.
- Type system can be used to build a noninterference static analysis.
  - $\Gamma \vdash e : l$ implies $\forall x \in e.\, \Gamma(x) \rightsquigarrow l$
  - $\Gamma \vdash S : l \; \texttt{cmd}$ implies $\forall (x := \ldots) \in S.\, l \rightsquigarrow \Gamma(x)$

Prepare for the upcoming

Do not forget to send me your workshop group:

- members list
- preference ordered list of the papers
- preferred presentation date

Exercise session 1:

- review your Python
- have a look at the Python's Taint Library paper

Exercise session 2:

- review your Java
- have a look at the Jif manual

Questions?

Questions?