

Semantics and Analysis of Programs for Security

Type systems

Thomas Jensen

Type systems

Plan :

- ▶ Introduction : what is a type?
- ▶ Monomorphic types. Simply-typed lambda calculus.
- ▶ Theorem: “Well-typed programs do not go wrong”.
- ▶ Type inference

References

- ▶ Benjamin C. Pierce : Types and Programming Languages. MIT Press, 2002
- ▶ Luca Cardelli. Type systems. Handbook of Computer Science and Engineering. CRC Press, 1996.

What is a type system

- ▶ Type theory was invented to eliminate certain logical paradoxes by classifying certain logical constructions as **non-sense**.
- ▶ In programming, types are used to determine what meaning to give to an expression like:

$$111000111 + 1$$

- ▶ Lots of languages have some kind of type systems: C, Ada, Caml, Java.
- ▶ Others, like LISP and Prolog, are **un-typed** languages (even though typed versions exist).
- ▶ A possible definition of a type system:

*A **type system** is a syntactic and efficient method for proving the absence of certain kinds of program behaviour, by classifying expressions according to the value they compute.*

What are types used for?

- 1 **Error detection.**
- 2 **Abstraction.** Facilitate the structuring of program into modules.
- 3 **Documentation.**
- 4 **Language safety.** Is the level of abstraction promised by a high-level language really ensured (eg. no low-level access to elements of an array). Caml is safe; C isn't.
- 5 **Performance.** Information about the type of an expression enables the compiler to generate more efficient code (optimal choice of numerical operators, elimination of certain run-time checks).

The simply-typed lambda calculus

We define a minimalistic programming language: the simply-typed lambda calculus extended with booleans and integers.

Syntax:

Expressions :

$$\begin{aligned} e ::= & \text{ true } \mid \text{ false } \mid \text{ if } e \text{ then } e \text{ else } e \\ & \mid \mathbf{0} \mid \text{ succ } e \mid \text{ pred } e \mid \text{ iszero } e \\ & \mid \mathbf{x} \\ & \mid \lambda \mathbf{x} : T. e \\ & \mid e e \end{aligned}$$

Types :

$$T ::= \text{ Nat } \mid \text{ Bool } \mid T \rightarrow T.$$

Operational semantics of the lambda calculus

The definition of the (small-step) operational semantics has two parts:

Values :

$v ::= nv \mid \text{true} \mid \text{false} \mid \lambda x : T.e$ **with** $nv ::= 0 \mid \text{succ } (nv)$

Transition relation:

$\text{pred } 0 \Rightarrow 0$ $\text{pred } (\text{succ } nv) \Rightarrow nv$

$\text{iszero } 0 \Rightarrow \text{true}$ $\text{iszero } (\text{succ } e) \Rightarrow \text{false}$

$\frac{e \Rightarrow e'}{\text{succ } e \Rightarrow \text{succ } e'}$ $\frac{e \Rightarrow e'}{\text{pred } e \Rightarrow \text{pred } e'}$ $\frac{e \Rightarrow e'}{\text{iszero } e \Rightarrow \text{iszero } e'}$

$\text{if true then } e_1 \text{ else } e_2 \Rightarrow e_1$ $\text{if false then } e_1 \text{ else } e_2 \Rightarrow e_2$

$\frac{e \Rightarrow e'}{\text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \text{if } e' \text{ then } e_1 \text{ else } e_2}$

$\frac{e_1 \Rightarrow e'_1}{e_1 e_2 \Rightarrow e'_1 e_2}$ $\frac{e_2 \Rightarrow e'_2}{v e_2 \Rightarrow v e'_2}$ $(\lambda x : T.e) v \Rightarrow e[v/x]$

Typing rules

Judgments of the form

$$A \vdash e : T$$

where A is an *environment* $\{x : T_1, y : T_2, \dots\}$.

$$\frac{A \vdash \text{true} : \text{Bool}}{A \vdash \text{pred } e : \text{Nat}} \quad \frac{A \vdash \text{false} : \text{Bool}}{A \vdash \text{iszero } e : \text{Bool}} \quad \frac{A \vdash 0 : \text{Nat}}{A \vdash \text{succ } e : \text{Nat}}$$
$$\frac{A \vdash e : \text{Bool} \quad A \vdash e_1 : T \quad A \vdash e_2 : T}{A \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T}$$
$$\frac{x : T \in A}{A \vdash x : T} \quad \frac{A \cup x : T_1 \vdash e : T_2}{A \vdash \lambda x : T_1. e : T_1 \rightarrow T_2}$$
$$\frac{A \vdash e_1 : T_1 \rightarrow T_2 \quad A \vdash e_2 : T_1}{A \vdash e_1 e_2 : T_2}$$

Examples

Example : $(\lambda x: \text{Bool} . x) \text{ true} .$

Exercise : Check that

$f: \text{Bool} \rightarrow \text{Bool} \vdash$
 $\lambda x: \text{Bool} . f(\text{if } x \text{ then false else } x): \text{Bool} \rightarrow \text{Bool}$

Type safety

We now prove that the type system allows to guarantee that the evaluation of a well-typed term does not **block** in a state where

- 1 the term is not a value (is not in normal form) and
- 2 no reduction rule apply to the term (eg. because we try to apply succ to a boolean).

Often summarised by the slogan “*Well-typed programs do not go wrong*”

More formally, we define

Safety = Progress + Invariance of types.

where

- ▶ **Progress** means that a well-typed term that is not in normal form can always be reduced.
- ▶ **Invariance** means that reduction preserves the type of a term.

Some properties of the type system

Lemma : Unicity of types.

Let A be an environment and let e be a term with all free variables defined in A . Then, there exists **at most** one type T such that $A \vdash e : T$.

Lemma : Inversion.

- 1 If $\text{true} : T$, then $T = \text{Bool}$.
- 2 If $\text{succ } e : T$, then $T = \text{Nat}$ and $e : \text{Nat}$.

Lemma : Canonical forms

- ▶ If v is a value of Nat , then v is a numerical value (defined by $nv ::= 0 \mid \text{succ } nv$).
- ▶ If v is a value of type Bool , then v is either `true` or `false`
- ▶ If v is a value of type $T_1 \rightarrow T_2$, then v is of form $\lambda x : T_1. e$

Progress

Theorem: Let e be a **closed** term (ie without free variables) of type T (ie $\vdash e : T$). Then,

either e is a value **or** there exists e' such that $e \Rightarrow e'$.

Proof: By induction on the derivation of the typing $\vdash e : T$. We do a case analysis on the last step in the derivation.

Case $\text{succ } e$. In this case, $T = \text{Nat}$ and we have that $e : \text{Nat}$. By induction hypothesis, two possibilities. Either e is a value and, by the Canonical Form lemma, an integer, in which case $\text{succ } e$ is also a value. Or $e \Rightarrow e'$ and by the semantic rules for \Rightarrow we have $\text{succ } e \Rightarrow \text{succ } e'$.

Case if is an exercise.

Progress (continued)

Case $e_1 e_2$. Three possibilities:

- 1 $e_1 \Rightarrow e'_1$. Then, $e_1 e_2 \Rightarrow e'_1 e_2$.
- 2 e_1 is a value, but $e_2 \Rightarrow e'_2$, Then, $e_1 e_2 \Rightarrow e_1 e'_2$.
- 3 e_1 and e_2 are values. By the Canonical Form lemma e_1 is of the form $\lambda x.e$ and the rule for β -reduction applies.

Substitution Lemma

This is a key property in the proof that evaluation preserves types. It states that types are invariant when we replace a variable with a term of the same type.

Lemma : If $A, x : S \vdash e : T$ and $A \vdash s : S$, then $A \vdash e[s/x] : T$.

Proof : By induction on the derivation of the judgment $A, x : S \vdash e : T$. As usual, we proceed by a case analysis on the last step in the derivation.

Case : $e = y$ inferred from the judgment $y : T \in (A, x : S)$. Two sub-cases to consider:

- ▶ $x = y$. Then $S = T$ and $e[s/x] = x[s/x] = s$, so we need to prove $A \vdash s : S$. But this typing is one of the hypotheses.
- ▶ $x \neq y$. Then, $y[s/x] = y$ and since $y : T \in A$, we have $A \vdash y[s/x] : T$.

Substitution Lemma (proof)

Case: $e = \lambda y : T_2.e_1$. In that case, $T = T_2 \rightarrow T_1$ and the pre-condition is $A, \mathbf{x} : S, y : T_2 \vdash e : T_1$.

But then we can also infer $A, y : T_2, \mathbf{x} : S \vdash e_1 : T_1$ (check!)

If $A \vdash s : S$ is derivable then so is $A, y : T_2 \vdash s : S$ (check!)

The induction hypothesis can now be applied to give

$$A, y : T_2 \vdash e_1[s/\mathbf{x}] : T_1$$

and, by the typing rule for abstraction, we infer

$$A \vdash \lambda y : T_2.e_1[s/\mathbf{x}] : T_2 \rightarrow T_1.$$

Now,

$$\lambda y : T_2.e_1[s/\mathbf{x}] = (\lambda y : T_2.e_1)[s/\mathbf{x}] = e[s/\mathbf{x}]$$

Case: Application $e = e_1 e_2$. **Exercise.**

Invariance (subject reduction)

Conservation of the type of an expression during its evaluation.

Often called “subject reduction”.

Theorem : If $A \vdash e : T$ and $e \Rightarrow e'$, then $A \vdash e' : T$.

Proof : By induction on the derivation of $A \vdash e : T$. So, we suppose that the theorem holds for all sub-derivations that have led to deduce $A \vdash e : T$. We now analyse each rule used in the last step of the derivation.

Case : $e = \text{true}$, $T = \text{Bool}$. Since true is a normal form, there is no e' such that $\text{true} \Rightarrow e'$. One of the hypotheses is thus false and the theorem is trivially true.

Case: $e = \text{succ } e_1$. The rule for succ yields that $T = \text{Nat}$ and $e_1 : \text{Nat}$. The only way to rewrite $\text{succ } e_1$ is if $e_1 \Rightarrow e'_1$ and hence, by induction hypothesis, we obtain that $A \vdash e'_1 : \text{Nat}$. By the typing rules for succ we deduce that $A \vdash \text{succ } e'_1 : \text{Nat}$ also.

Case: $e = \text{if } b \text{ then } e_1 \text{ else } e_2$ is an exercise.

Invariance (proof)

Case : Application $e = e_1 e_2$. Then $A \vdash e_1 : T_1 \rightarrow T$ and $A \vdash e_2 : T_1$. If $e_1 e_2 \Rightarrow e'$, then **either** $e_1 \Rightarrow e'_1$ or $e_2 \Rightarrow e'_2$ (two trivial cases) **or** $e_1 = \lambda \mathbf{x} : T_1.s$ and

$$e' = s[e_2 / \mathbf{x}]$$

by a beta-reduction. Now, if

$$A \vdash \lambda \mathbf{x} : T_1.s : T_1 \rightarrow T,$$

then there is a deduction of

$$A, \mathbf{x} : T_1 \vdash s : T.$$

Together with the fact that $A \vdash e_2 : T_1$, the Substitution Lemma gives that

$$A \vdash s[e_2 / \mathbf{x}] : T.$$

Case : Abstraction. Trivial (why?)

Extensions : products and sums

Most languages have constructions for building complex data structures.

Pairs.

Expressions $e ::= \dots \mid (e_1, e_2) \mid e.1 \mid e.2$

Values $v ::= \dots \mid (v_1, v_2)$

Types $T ::= \dots \mid T_1 \times T_2.$

Evaluation

$(v_1, v_2).1 \Rightarrow v_1$ $(v_1, v_2).2 \Rightarrow v_2$ $\frac{e_1 \Rightarrow e'_1}{e_1.1 \Rightarrow e'_1.1}$ \dots $\frac{e_1 \Rightarrow e'_1}{(e_1, e_2) \Rightarrow (e'_1, e_2)}$ \dots

Typing rules

$$\frac{A \vdash e_1 : T_1 \quad A \vdash e_2 : T_2}{A \vdash (e_1, e_2) : T_1 \times T_2}$$

$$\frac{A \vdash e : T_1 \times T_2}{e.1 : T_1}$$

$$\frac{A \vdash e : T_1 \times T_2}{e.2 : T_2}$$

Extensions : products and sums

Sums

Example :

```
PhysicalAddress = {firstlast : String, addr : String}
```

```
VirtualAddress = {name : String, email : String}
```

```
Addr = PhysicalAddress + VirtualAddress
```

Expressions $e ::= \dots \mid \text{inl } e \mid \text{inr } e \mid \text{case } e \text{ of inl } x \triangleright e [] \text{inr } x \triangleright e.$

Values $v ::= \dots \mid \text{inl } v \mid \text{inr } v$

Types $T ::= \dots \mid T_1 + T_2.$

Evaluation

case inl v_0 of inl $x_1 \triangleright e_1$ [] inr $x_2 \triangleright e_2 \Rightarrow e_1[v_0/x_1]$

case inr v_0 of inl $x_1 \triangleright e_1$ [] inr $x_2 \triangleright e_2 \Rightarrow e_2[v_0/x_2]$

$$\frac{e \Rightarrow e'}{\text{case } e \text{ of inl } x \triangleright e \text{ [] inr } x \triangleright e \Rightarrow \text{case } e' \text{ of inl } x \triangleright e \text{ [] inr } x \triangleright e}$$
$$\frac{e \Rightarrow e'}{\text{inr } e \Rightarrow \text{inr } e'} \quad \frac{e \Rightarrow e'}{\text{inr } e \Rightarrow \text{inr } e'}$$

Typing rules

$$\frac{A \vdash e : T_1}{A \vdash \text{inl } e_1 : T_1 + T_2} \quad \frac{A \vdash e : T_2}{\text{inr } e : T_1 + T_2}$$
$$\frac{A \vdash e : T_1 + T_2 \quad A, x_1 : T_1 \vdash e_1 : T \quad A, x_2 : T_2 \vdash e_2 : T}{A \vdash \text{case } e \text{ of inl } x_1 \triangleright e_1 \text{ [] inr } x_2 \triangleright e_2 : T}$$

NB : With sum types, we no longer have type unicity

Explicit type declarations

Expressions $e ::= \dots \mid e \text{ as } T$

Values $v ::= \dots$

Types $T ::= \dots$

Evaluation

$$v \text{ as } T \Rightarrow v \quad \frac{e \Rightarrow e'}{e \text{ as } T \Rightarrow e' \text{ as } T}$$

Typing rules

$$\frac{A \vdash e : T}{A \vdash e \text{ as } T : T}$$

In particular useful for [documentation](#) and program formatting.

In languages with [sub-typing](#) there are similar operations for checking and modifying a type (the “**casting**” of a type).

Typing recursive functions

Examples :

```
letrec fac x = if x < 2 then 1 else x * fac(x-1)
fac = fix λ f . λ x . if x < 2 then 1 else x * f(x-1)
```

Expressions $e ::= \dots \mid \mathbf{fix} e$

Values $v ::= \dots$

Types $T ::= \dots$

Evaluation

$$\mathbf{fix} (\lambda \mathbf{x} : T.e) \Rightarrow e[\mathbf{fix} (\lambda \mathbf{x} : T.e) / \mathbf{x}] \qquad \frac{e \Rightarrow e'}{\mathbf{fix} e \Rightarrow \mathbf{fix} e'}$$

Typing rules

$$\frac{A \vdash e : T \rightarrow T}{A \vdash \mathbf{fix} e : T}$$

Type inference

We add an infinite set of **type variables** X, Y, Z to the set of types

$$T ::= \text{Nat} \mid \text{Bool} \mid \mathbf{X} \mid T \rightarrow T$$

Definition : A substitution of types is a finite function σ from type variables to types.

A substitution extends to types in the obvious way.

$$\sigma(T) = \left\{ \begin{array}{l} \end{array} \right.$$

With this notion, we can state the type inference problem formally.

Definition : Let A be an environment and e a term. A *solution* to the typing problem (A, e) is a type T and a substitution σ such that

$$\sigma(A) \vdash \sigma(e) : T$$

Typing by constraint solving

Define a typing relation

$$A \vdash e : T, C$$

where C is a set of constraints on the types in A, e, T .

Intuition :

e has type T under the hypothesis A if C is satisfied.

A set of constraints C is of form $\{S_i = T_i\}_{i \in I}$.

A substitution σ **satisfies** C if $\sigma(S_i) = \sigma(T_i)$ for all $i \in I$.

Constraint-based typing rules

$$A \vdash \text{true} : \text{Bool}, \emptyset \quad A \vdash \text{false} : \text{Bool}, \emptyset \quad A \vdash 0 : \text{Nat}, \emptyset$$
$$\frac{A \vdash e : T, C}{A \vdash \text{pred } e : \text{Nat}, C \cup \{T = \text{Nat}\}} \quad \frac{A \vdash e : T, C}{A \vdash \text{iszero } e : \text{Bool}, C \cup \{T = \text{Nat}\}}$$
$$\frac{x : T \in A}{A \vdash x : T, \emptyset} \quad \frac{A \vdash e : T, C}{A \vdash \text{succ } e : \text{Nat}, C \cup \{T = \text{Nat}\}}$$
$$\frac{A \vdash e : \text{Bool}, C_1 \quad A \vdash e_1 : T, C_2 \quad A \vdash e_2 : T, C_3}{A \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T, C_1 \cup C_2 \cup C_3}$$
$$\frac{A, x : T_1 \vdash e : T_2, C}{A \vdash \lambda x : T_1. e : T_1 \rightarrow T_2, C} \quad \frac{A \vdash e_1 : T_1, C_1 \quad A \vdash e_2 : T_2, C_2}{A \vdash e_1 e_2 : X, C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow X\}}$$

Condition : Variables in C_1, C_2, C_3 must be distinct.

Exercise : determine T, C in $\vdash \lambda x : X. \lambda y : Y. \lambda z : Z. (x z) (y z) : T, C$

Correctness and completeness

The constraint-based system infers a relation

$$A \vdash e : S, C$$

What is the link with the original type system?

Definition : A **solution** to (A, e, S, C) is a pair (σ, T) such that

$$\sigma \text{ satisfies } C \quad \text{and} \quad \sigma(S) = T$$

Two results will prove the equivalence of the systems:

- 1 Every solution to (A, e, S, C) is also a solution to $A \vdash e : ?$ (correctness)
- 2 Every solution to $A \vdash e : ?$ can be extended to a solution to (A, e, S, C) by giving values to the type variables that have been introduced during type inference (Completeness).

Correctness

Theorem : Suppose $A \vdash e : S, C$. If (σ, T) is a solution to (A, e, S, C) , then it is also a solution to (A, e) .

Proof: Induction on the derivation of $A \vdash e : S, C$, by doing a case analysis on the last rule used in the derivation.

Case $e = x$. Then $x : S \in A$ and $C = \emptyset$. If (σ, T) is a solution, then $\sigma S = T$, so $x : T \in \sigma(A)$ and we can infer that $\sigma(A) \vdash x : T$.

Case $t = \lambda x.e_2$. Then, $S = T_1 \rightarrow S_2$ and $A, x : T_1 \vdash e_2 : S_2, C$.

If (σ, T) is a solution, then σ satisfies C and $T = \sigma(S) = \sigma(T_1) \rightarrow \sigma(S_2)$.
As a consequence, $(\sigma, \sigma(S_2))$ is a solution to $(A, x : T_1), e_2, \sigma(S_2), C$.

By induction hypothesis, $(\sigma, \sigma(S_2))$ is also a solution to $((A, x : T_1), e_2)$, so $\sigma A, x : \sigma T_1 \vdash \sigma e_2 : \sigma(S_2)$. By the typing rule for λ terms

$$\sigma A \vdash \lambda x.e_2 : \sigma T_1 \rightarrow \sigma(S_2) = \sigma(T_1 \rightarrow S_2) = \sigma(S) = T.$$

Completeness

The Completeness Theorem is more complicated because of intermediate type variables, introduced during type inference. The book-keeping for these variables complicates the proof so we will just state the theorem. See the book by Pierce (section 22.3.7) for a detailed proof.

Theorem (Completeness.) Suppose that $A \vdash e : S, C$ can be inferred using the intermediate type variables \mathcal{X} .

If (σ, T) is a solution to (A, e) **and** if $Dom(\sigma) \cap \mathcal{X} = \emptyset$, then there exists a solution (σ', T) of (A, e, S, C) such that $\sigma'(\mathbf{x}) = \sigma(\mathbf{x})$ for all $\mathbf{x} \in Dom(\sigma)$.

Unification : solving the constraints

The constraints can be solved by a procedure called **unification**.

```
unify(C) = if C = { } then [] else
  let { S = T } ∪ C' = C in
    if S = T
      then unify(C')
    else if S = X and X ∉ FV(T)
      then unify(C' [T/X]) ∘ [X ↦ T]
    else if T = X and X ∉ FV(S)
      then unify(C' [S/X]) ∘ [X ↦ S]
    else if S = S1 -> S2 and T = T1 -> T2
      then unify(C' ∪ {S1 = T1, S2 = T2})
    else fail
```

Unification : main theorem

Theorem : The `unify` algorithm always terminates and finds the most general unifier if it exists.

Exercise : **Unify**

$$\{X = \text{Nat} , Y = X \rightarrow X\}$$
$$\{X \rightarrow Y = Y \rightarrow Z, Z = U \rightarrow W\}$$

Principal types

Question : Do we always get the best typing?

One substitution σ is more general than τ if τ can be obtained from σ :
 $\tau = \gamma \circ \sigma$.

A *principal solution* of (A, e, S, C) is a solution (σ, T) such that for all other solutions (σ', T') , σ is more general than σ' .

In that case, T is the **principal type**.

Theorem : (Principal type) If the typing problem (A, e, S, C) has a solution, then it has a **principal** solution

Proof : Immediate, as soon as we have that the unification algorithm computes the most general unifier.