

Compilation and Program Analysis (#4) :

Types, and Typing MiniWhile

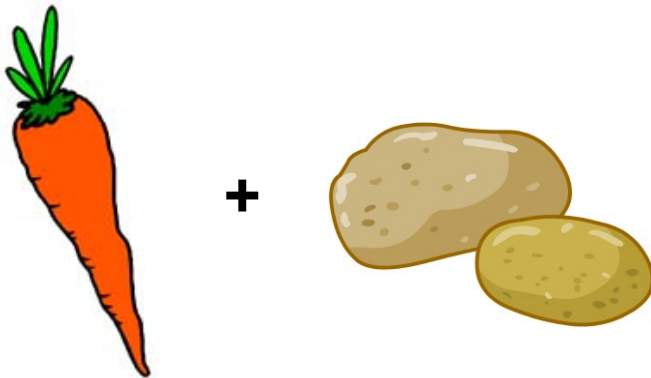
Ludovic Henrio

Master 1, ENS de Lyon et Dpt Info, Lyon1

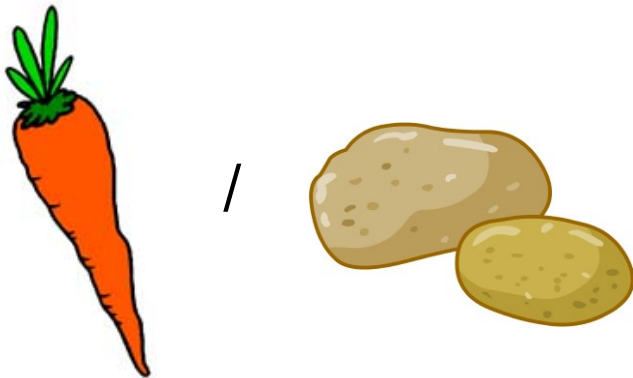
2021-2022



Typing



Typing



Typing

If you write: `"5" + 37`
what do you want to obtain

- a compilation error? (OCaml)
- an exec error? (Python)
- the int 42? (Visual Basic, PHP)
- the string `"537"`? (Java)
- anything else?

and what about `37 / "5" ?`

Typing

When is

$e1 + e2$

legal, and what are the semantic actions to perform ?

► Typing: an analysis that gives a type to each subexpression, and reject incoherent programs.

When

- Dynamic typing (during exec): Lisp, PHP, Python
 - Static typing (at compile time): C, Java, OCaml
- Here: the second one.

Slogan

well typed programs do not go wrong

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
- 3 Type Safety
- 4 A bit of implementation

Typing objectives

- Should be **decidable**.
- It should reject programs like `(1 2)` in OCaml, or `1+"toto"` in C before an actual error in the evaluation of the expression: this is **safety**.

The type system is related to the kind of error to be detected:
operations on basic types / method invocation (message not understood) / correct synchronisation (e.g. session types) in concurrent programs / ...

- The type system should be expressive enough and not reject too many programs. (**expressivity**)

Principle

All sub-expressions of the program must be given a type

```
fun (x : int) → let (y : int) = (+ :)((x : int), (1 : int)) : int × int in
```

What does the programmer write?

- The type of all sub-expressions (like above) easy to verify, but tedious for the programmer
- Annotate only variable declarations (Pascal, C, Java, ...)

```
fun (x : int) → let (y : int) = +(x, 1) in y
```

- Only annotate function parameters

```
fun (x : int) → let y = +(x, 1) in y
```

- Annotate nothing: complete inference : Ocaml, Haskell, ...

Properties

- correction: “yes” implies the program is well typed.
- completeness: the converse.

(optional)

- principality : The most general type is computed.

Typing judgement

We will define how to compute **typing judgements** denoted by:

$$\Gamma \vdash e : \tau$$

and means “in environment Γ , expression e has type τ ”

► Γ associates a type $\Gamma(x)$ to all free variables x in e .

Safety = well typed programs do not go wrong

In general a type-safety property looks like this:

Theorem (Safety)

If $\emptyset \vdash e : \tau$, then the reduction of e is infinite or terminates with a value.

Typing Safety

In general, a type-safety proof is based on two lemmas:

Lemme (progression)

If $\emptyset \vdash e : \tau$, then e is a value or there exists e' such that $e \rightarrow e'$.

Lemme (preservation)

If $\emptyset \vdash e : \tau$ and $e \rightarrow e'$ then $\emptyset \vdash e' : \tau$.

This works almost the same for small-step and big-step.

What is a good output for a type-checker?

- We do not want:

`failwith "typing error"`

the origin of the problem should be clearly stated

- We keep the types for next phases.

In practice

- Input: Trees are decorated by source code lines.
- Output: Trees are decorated by types.

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - More advanced typing features
- 3 Type Safety
- 4 A bit of implementation

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - More advanced typing features
- 3 Type Safety
 - Type safety for expressions
 - Recall the typing system for mini-while
 - Safety
- 4 A bit of implementation

Mini-While Syntax

Expressions (with boolean expression to make typing interesting):

$e ::= c$	<i>constant</i>
x	<i>variable</i>
$e + e$	<i>addition</i>
$e \times e$	<i>multiplication</i>
$e < e$	<i>boolean expression</i>
...	

Mini-while:

$S(Smt) ::= x := expr$	<i>assign</i>
$skip$	<i>do nothing</i>
$S_1; S_2$	<i>sequence</i>
$\text{if } e \text{ then } S_1 \text{ else } S_2$	<i>test</i>
$\text{while } e \text{ do } S \text{ done}$	<i>loop</i>

Typing rules for expr

Here types are basic types: `Int|Bool`

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \text{int}} \quad (\text{or tt: bool, } \dots)$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 < e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \wedge e_2 : \text{bool}} \quad \dots$$

Example

show that $(x + 42)$ is well typed in an environment where $[x \mapsto int]$

Typing rules for statements: $\Gamma \vdash S$

A statement S is well-typed (there is no type for statements)
on board!

Typing: an example

Considering $\Gamma = \{x \mapsto int\}$, prove that the given sequence of instructions is well typed:

$x = 3 ;$

$x = x+9 ;$

on board!

Problem: how to define Γ in mini-while? (1/2)

Possible solution: programs declare variables:

$$\begin{array}{ll}
 P ::= D; S & \text{program} \\
 D ::= \text{var } x : \tau \mid D; D & \text{Variable declaration}
 \end{array}$$

Suppose the operational semantics discard the variable declaration for now but Γ is defined according to declarations:
 for a given program $D; S$: $\Gamma(x) = \tau \iff \text{var } x : \tau \in D$

Problem: how to define Γ in mini-while? (2/2)

Γ can be defined as a typing rule for programs.

From declarations we infer $\Gamma : Var \rightarrow Basetype$ with the two following rules:

$$\overline{var\ x : t \rightarrow_d [x \mapsto t]}$$

$$\frac{D_1 \rightarrow_d \Gamma_1 \quad D_2 \rightarrow_d \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D_1; D_2 \rightarrow_d \Gamma_1 \cup \Gamma_2}$$

Typing of programs can be defined as follows:

$$\frac{D \rightarrow_d \Gamma \quad \Gamma \vdash S}{\emptyset \vdash D; S}$$

Typing a “runtime configuration”

In mini-while, initial state of big step and almost all states of SOS are not just statements, they are pairs: (statement, store). To reason on types at runtime we first need to type runtime configurations:

Definition (Configuration typing)

$$\Gamma \vdash (S, \sigma) \iff (\Gamma \vdash S \wedge \forall x. \emptyset \vdash \sigma(x) : \tau \iff \Gamma(x) = \tau)$$

Notes:

- $\sigma(x)$ is a value: no Γ is needed to type it
- last part somehow says Γ and σ agree on the type of variables.

Example continued

What is the full program corresponding to the previous example?

```
x = 3 ;  
x = x+9 ;
```

What is the initial σ for this program? Does it agree with Γ ?
see later ...

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - More advanced typing features
- 3 Type Safety
 - Type safety for expressions
 - Recall the typing system for mini-while
 - Safety
- 4 A bit of implementation

Hybrid expressions

What if we have $1.2 + 42$?

- reject?
- compute a float!

► This is **type coercion**. We will see how to implement it during a lab.

► It requires a very local form of type inference.

More complex expressions

What if we have types `pointer of bool` or `array of int`?
We might want to check equivalence (for addition ...).

► This is called **structural equivalence** (see Dragon Book, “type equivalence”). This is solved by a basic graph traversal checking that each element are equivalent/compatible.

Sub-typing

- A type can be more precise than another one, e.g.

$$int <: num$$

- Need additional rule to use sub-typing:

$$\frac{e : \tau \quad \tau <: \tau'}{e : \tau'}$$

- Sometimes, rule to compose sub-types, e.g. functions or parametric types

$$\frac{e : List[\tau] \quad \tau <: \tau'}{e : List[\tau']}$$

Note: subtyping is heavily used in OOP

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
- 3 Type Safety
 - Type safety for expressions
 - Recall the typing system for mini-while
 - Safety
- 4 A bit of implementation

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - More advanced typing features
- 3 Type Safety
 - Type safety for expressions
 - Recall the typing system for mini-while
 - Safety
- 4 A bit of implementation

One-step type safety

Theorem (Type correctness for big-step semantics)

If $\emptyset \vdash e : \tau$ and $e \longrightarrow v$ then the value v is of the right type $\emptyset \vdash v : \tau$.

Another one-step reduction is expression evaluation.

Suppose for now Γ and σ agree on types and we prove correctness of expression types as follows:

Prove the following theorem (variant of type correctness)

*Suppose $\forall x \in \text{vars}(e). \emptyset \vdash \sigma(x) : \tau \iff \Gamma(x) = \tau$
prove type correctness for $\text{Val}(e, \sigma)$, i.e.:*

$\Gamma \vdash e : \tau \implies \emptyset \vdash \text{Val}(e, \sigma) : \tau$

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - More advanced typing features
- 3 Type Safety
 - Type safety for expressions
 - Recall the typing system for mini-while
 - Safety
- 4 A bit of implementation

Typing recap 1/2

$P ::= D; S$ program

$D ::= \text{var } x : t$ type declaration

From declarations we infer $\Gamma : Var \rightarrow Basetype$ with the two following rules:

$$\frac{}{\text{var } x : t \rightarrow_d [x \mapsto t]}$$

$$\frac{D_1 \rightarrow_d \Gamma_1 \quad D_2 \rightarrow_d \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D_1; D_2 \rightarrow_d \Gamma_1 \cup \Gamma_2}$$

Typing recap 2/2

Then a typing judgment for expressions is $\Gamma \vdash e : \tau \in Basetype$.
Statements have no type and judgement is: $\Gamma \vdash S$.

$$\frac{D \rightarrow_d \Gamma \quad \Gamma \vdash S}{\emptyset \vdash D; S} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \qquad \Gamma \vdash x : \Gamma(x)$$

$$\frac{c \in \mathbf{Z}}{c : \text{int}} \qquad \frac{b \in \mathbb{B}}{c : \text{bool}} \qquad \frac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1; S_2} \qquad \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e}$$

$$\frac{\Gamma \vdash b : \text{bool} \quad \Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash \text{if } b \text{ then } S_1 \text{ else } S_2} \qquad \frac{\Gamma \vdash b : \text{bool} \quad \Gamma \vdash S}{\Gamma \vdash \text{while } b \text{ do } S \text{ done}}$$

Typing configurations:

$$\Gamma \vdash (S, \sigma) \iff (\Gamma \vdash S \wedge \forall x. \emptyset \vdash \sigma(x) : \tau \iff \Gamma(x) = \tau)$$

Before getting into technical proof

What “wrong behaviours” are prevented by our type-system?

- ▷ We will try to prove this.

What wrong behaviours are still here?

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - More advanced typing features
- 3 Type Safety
 - Type safety for expressions
 - Recall the typing system for mini-while
 - **Safety**
- 4 A bit of implementation

Safety = well typed programs do not go wrong

In case of a small-step semantics the proof that “well typed programs do not go wrong” relies on two lemmas:

Well-type programs run without error

Lemma (progression for mini-while)

*If $\Gamma \vdash (S, \sigma)$, then there exists S', σ' such that $(S, \sigma) \Rightarrow (S', \sigma')$
OR there exists σ' such that $(S, \sigma) \Rightarrow \sigma'$.*

Note: (S, σ) cannot be a final configuration.

... and remain well-typed

Lemma (preservation)

If $\Gamma \vdash (S, \sigma)$ and $(S, \sigma) \Rightarrow (S', \sigma')$ then $\Gamma \vdash (S', \sigma')$.

Note that Γ never changes (defined by declarations)

Proofs! (recall the property for expression evaluation)

Initial Configuration

Problem: how do we start execution?

Until now (S, \emptyset) was a good starting configuration.

But Γ and \emptyset do not agree on the typing

what is a good initial σ ?

Initialisation or consider that $\sigma(x)$ is fine?

Discussion + refer to practical session.

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
- 3 Type Safety
- 4 A bit of implementation

Principle

- Gamma is constructed with lexing information or parsing (variable declaration with types).
- Rules are semantic actions. The semantic actions are responsible for the evaluation order, as well as typing errors.

Type Checking V1 : Visitor

MuTypingVisitor.py

```
# now visit expr
```

```
def visitAtomExpr(self, ctx):  
    return self.visit(ctx.atom())
```

```
def visitOrExpr(self, ctx):  
    lvaltype = self.visit(ctx.expr(0))  
    rvaltype = self.visit(ctx.expr(1))  
    if (BaseType.Boolean == lvaltype) and (BaseType.Boolean == rvaltype):  
        return BaseType.Boolean  
    else:  
        self._raise(ctx, 'boolean operands', lvaltype, rvaltype)
```

In practice for mini-C (lab sessions)

No annotation is added to the AST (everything is int or bool, no ambiguity)

We can create associating type to variables, directly from parsing

Conclusion

We have seen:

- The properties and principle of static typing
- A type system for miniml
- A type system for mini-while
- Type safety and how to prove it — miniml and mini while
- discussion on variable declaration and initialisation

Further discussions not really covered in the course:

- Typing functions (later in the course)
- More complex (i.e. real life) type system: sub-typing, objects, functions and sub-typing
- There exist very rich type systems (research on type system), e.g. behavioural types, ownership types, ...