

Compilation and Program Analysis (#4) :

Types, Typing

Laure Gonnord

<http://laure.gonnord.org/pro/teaching/capM1.html>

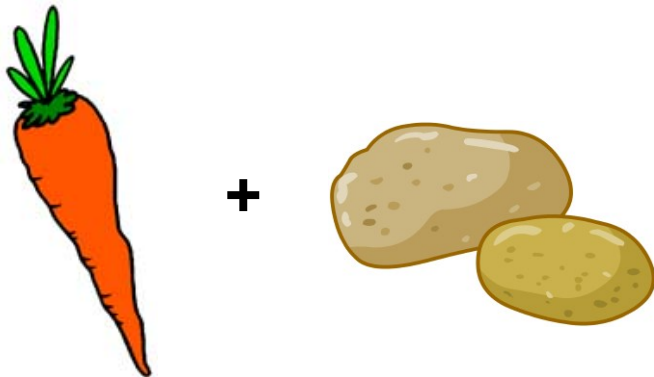
Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

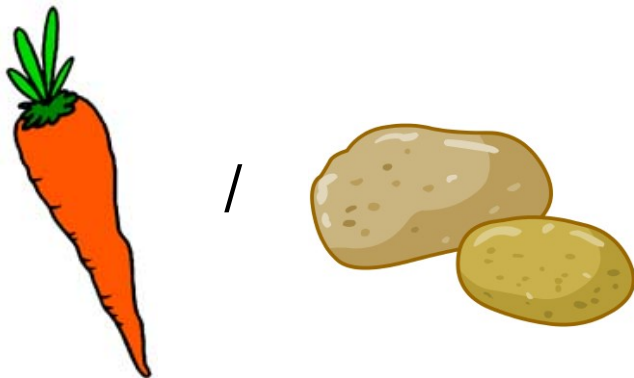
2018-2019



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

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 should be expressive enough and not reject too many programs. (**expressivity**)

Several solutions

- All sub-expressions are annotated by a type

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

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
```

- Do 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.

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
- 3 Functional languages (ML)

In practice

- We do not want :

```
failwith "typing error"
```

the origin of the problem should be clearly stated

- We keep the types for next phases.

Typing 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
 - A bit of implementation (for expr)
 - Typing functions
- 3 Functional languages (ML)

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - A bit of implementation (for expr)
 - Typing functions
- 3 Functional languages (ML)
 - Monomorphic typing of mini-ML
 - Implementation
 - Parametric polymorphism for ML

Mini-While Syntax

Expressions :

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

Mini-while :

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

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 .

Here types are basic types : Int|String|Bool

Typing rules for expr

$$\overline{\Gamma \vdash x : \Gamma(x)} \quad \overline{\Gamma \vdash n : \text{int}} \text{ (or bool, ...)}$$

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

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 Lab 4.

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.

Typing rules for statements

on board !

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - A bit of implementation (for expr)
 - Typing functions
- 3 Functional languages (ML)
 - Monomorphic typing of mini-ML
 - Implementation
 - Parametric polymorphism for ML

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

```
def visitAndExpr(self, ctx):
    # BEGIN CUT
    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)
    # END CUT
    # UNCOMMENT raise NotImplementedError()

def visitEqualityExpr(self, ctx):
    # BEGIN CUT
```


Type Checking V2 : from AST to decorated ASTs

Idea

- Generate an AST for the parsed file.
- Decorate with types with a tree traversal.

AST type in Python

Ast.py

```
""" Expressions """
class BinOp(Expression):
    def __init__(self, left, right):
        super(Expression, self).__init__()
        self.left = left
        self.right = right

class AddOp(BinOp):
    def eval(self):
        return self.left.eval() + self.right.eval()
```

AST generation in Python

This AST is generated with the ANTLR visitor from our grammar :

MyAritVisitor.py

```
def visitAdditiveExpr(self, ctx):
    leftval = self.visit(ctx.expr(0))
    rightval = self.visit(ctx.expr(1))
    if ( self.visit(ctx.pmpop()) == '+' ): #see lab for a
        better way to match ops
        return AddOp(left=leftval, right=rightval)
    else:
        return SubOp(left=leftval, right=rightval)
```

- ▶ Types can be computed and stored inside this visitor ! (see Labs 4 and 5)

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - A bit of implementation (for expr)
 - **Typing functions**
- 3 Functional languages (ML)
 - Monomorphic typing of mini-ML
 - Implementation
 - Parametric polymorphism for ML

Later

In the course dedicated to functions in mini-while.

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
- 3 Functional languages (ML)
 - Monomorphic typing of mini-ML
 - Implementation
 - Parametric polymorphism for ML

Mini-ml typing

- monomorphic typing
- polymorphism, type inference

Mini-ML, syntax recall

$e ::= x$	ident
c	constant (1, 2, ..., <i>true</i> , ...)
op	primitive (+, ×, <i>fst</i> , ...)
$\text{fun } x \rightarrow e$	function
$e e$	application
(e, e)	pair
$\text{let } x = e \text{ in } e$	local binding

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - A bit of implementation (for expr)
 - Typing functions
- 3 Functional languages (ML)
 - **Monomorphic typing of mini-ML**
 - Implementation
 - Parametric polymorphism for ML

Monomorphic typing of mini-ML

Abstract syntax for (simple) types :

$$\begin{array}{ll} \tau ::= \text{int} \mid \text{bool} \mid \dots & \textit{base type} \\ | \tau \rightarrow \tau & \textit{function type} \\ | \tau \times \tau & \textit{type} \end{array}$$

Typing rules for mini-ml

$$\overline{\Gamma \vdash x : \Gamma(x)} \quad \overline{\Gamma \vdash n : \text{int}} \cdots \quad \overline{\Gamma \vdash + : \text{int} \times \text{int} \rightarrow \text{int}} \cdots$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

$\Gamma + x : \tau$ is Γ' defined by $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ for all $y \neq x$.

Example

$$\frac{
 \frac{
 \frac{
 \vdots
 }{
 x : \text{int} \vdash (x, 1) : \text{int} \times \text{int}
 }
 }{
 x : \text{int} \vdash +(x, 1) : \text{int}
 }
 }{
 \emptyset \vdash \text{fun } x \rightarrow +(x, 1) : \text{int} \rightarrow \text{int}
 }
 \quad
 \frac{
 \frac{
 \dots \vdash f : \text{int} \rightarrow \text{int}
 }{
 f : \text{int} \rightarrow \text{int} \vdash f \ 2 : \text{int}
 }
 \quad
 \frac{
 \dots \vdash 2 : \text{int}
 }{
 }
 }{
 }
 }{
 \emptyset \vdash \text{let } f = \text{fun } x \rightarrow +(x, 1) \text{ in } f \ 2 : \text{int}
 }$$

Non typable expressions

We cannot type $1\ 2$:

$$\frac{\Gamma \vdash 1 : \tau' \rightarrow \tau \quad \Gamma \vdash 2 : \tau'}{\Gamma \vdash 1\ 2 : \tau}$$

nor $\text{fun } x \rightarrow x\ x$

$$\frac{\frac{\Gamma + x : \tau_1 \vdash x : \tau_3 \rightarrow \tau_2 \quad \Gamma + x : \tau_1 \vdash x : \tau_3}{\Gamma + x : \tau_1 \vdash x\ x : \tau_2}}{\Gamma \vdash \text{fun } x \rightarrow x\ x : \tau_1 \rightarrow \tau_2}$$

because $\tau_1 = \tau_1 \rightarrow \tau_2$ has no solution (types are finite)

Several possible types

We can show :

$$\emptyset \vdash \text{fun } x \rightarrow x : \text{int} \rightarrow \text{int}$$

but also :

$$\emptyset \vdash \text{fun } x \rightarrow x : \text{bool} \rightarrow \text{bool}$$

This is **not polymorphism** : for each occurrence of $\text{fun } x \rightarrow x$ we have to *choose* a type.

Several possible types 2/2

The term `let f = fun x → x in (f 1, f true)` is not typable because there is no τ such that :

$$f : \tau \rightarrow \tau \vdash (f\ 1, f\ \mathbf{true}) : \tau_1 \times \tau_2$$

nevertheless :

$$((\mathbf{fun}\ x \rightarrow x)\ (\mathbf{fun}\ x \rightarrow x))\ 42$$

is typable (exercise !)

Primitives : pb

To give a type to `fst`, we should choose between :

`int × int → int`

`int × bool → int`

`bool × int → bool`

`(int → int) × int → int → int`

etc.

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - A bit of implementation (for expr)
 - Typing functions
- 3 Functional languages (ML)
 - Monomorphic typing of mini-ML
 - **Implementation**
 - Parametric polymorphism for ML

Typing functions of mini-ML

How to we find the type to give to x when typing `fun $x \rightarrow e$` ?

Implementation of the simple typing

Rules of the game :

- functions parameters are annotated.
- program in Ocaml.

Implementation 1/4

Abstract Syntax for types

```
type typ =  
  | Tint  
  | Tarrow   of typ * typ  
  | Tproduct of typ * typ
```

Implementation 2/4

Abstract Syntax for expr - functions are annotated

```
type expression =  
  | Var    of string  
  | Const of int  
  | Op     of string  
  | Fun    of string * typ * expression (* only change here! *)  
  | App    of expression * expression  
  | Pair   of expression * expression  
  | Let    of string * expression * expression
```

The environment (Γ) is a persistent datastructure :

```
module Smap = Map.Make(String)  
type env = typ Smap.t
```

Implementation 3/4

Typing expr

```
let rec type_expr env = function
  | Const _ -> Tint
  | Var x -> Smap.find x env
  | Op "+" -> Tarrow (Tproduct (Tint, Tint), Tint)
  | Pair (e1, e2) ->
      Tproduct (type_expr env e1, type_expr env e2)
```

Local var : type is computed

```
| Let (x, e1, e2) ->
    type_expr (Smap.add x (type_expr env e1) env) e2
```

Implementation 4/4

Fun : var type is given !

```
| Fun (x, ty, e) →  
  Tarrow (ty, type_expr (Smap.add x ty env) e)
```

Typing verification when apply :

Apply type check !

```
| App (e1, e2) → begin match type_expr env e1 with  
  | Tarrow (ty2, ty) →  
    if type_expr env e2 = ty2 then ty  
    else failwith "error: wrong type of argument"  
  | _ →  
    failwith "error: function required"  
end
```

Implem : examples

```
# type_expr
  (Let ("f",
      Fun ("x", Tint, App (Op "+", Pair (Var "x", Const 1))),
      App (Var "f", Const 2)));;
```

— : typ = Tint

```
# type_expr (Fun ("x", Tint, App (Var "x", Var "x")));;
```

Exception: Failure "error: function expected".

```
# type_expr (App (App (Op "+", Const 1), Const 2));;
```

Exception: Failure "error: argument of wrong type"

Typing safety

well typed programs do not go wrong

Safety

For mini-ml, we show the adequacy of the typing process wrt the reduction semantics. (small-steps, not in the course)

Theorem (Safety)

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

Typing Safety

The 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 $\emptyset \vdash e' : \tau$.

- 1 Generalities about typing
- 2 Imperative languages (C, Mini-While)
 - Simple Type Checking for mini-while
 - A bit of implementation (for expr)
 - Typing functions
- 3 Functional languages (ML)
 - Monomorphic typing of mini-ML
 - Implementation
 - **Parametric polymorphism for ML**

Polymorphism for mini-ML

Give a unique type to `fun x → x in`

```
let  $f = \text{fun } x \rightarrow x \text{ in } \dots$ 
```

is a too big constraint. Moreover, we want to give “several types” to `fst` or `snd`.

► **Parametric polymorphism**

Parametric polymorphism

Types' abstract syntax is now :

$\tau ::=$	<code>int</code> <code>bool</code> ...	<i>base types</i>
	$\tau \rightarrow \tau$	<i>function type</i>
	$\tau \times \tau$	<i>pair</i>
	α	<i>variable type</i>
	$\forall \alpha. \tau$	<i>polymorphic type</i>

F-system

Same rules plus :

$$\frac{\Gamma \vdash e : \tau \quad \alpha \notin \text{FV}(\Gamma)}{\Gamma \vdash e : \forall \alpha. \tau}$$

and

$$\frac{\Gamma \vdash e : \forall \alpha. \tau}{\Gamma \vdash e : \tau[\alpha \leftarrow \tau']}$$

$\text{FV}(\Gamma) ==$ free variables of Γ

► F-System (J.-Y. Girard / J. C. Reynolds)

Example

$$\frac{\frac{x : \alpha \vdash x : \alpha}{\vdash \mathbf{fun} x \rightarrow x : \alpha \rightarrow \alpha} \quad \frac{\frac{\dots \vdash f : \forall \alpha. \alpha \rightarrow \alpha}{\dots \vdash f : \mathbf{int} \rightarrow \mathbf{int}} \quad \vdots}{\dots \vdash f \mathbf{1} : \mathbf{int}} \quad \frac{\vdots}{\dots \vdash f \mathbf{true} : \mathbf{bool}}}{\frac{\vdash \mathbf{fun} x \rightarrow x : \forall \alpha. \alpha \rightarrow \alpha \quad f : \forall \alpha. \alpha \rightarrow \alpha \vdash (f \mathbf{1}, f \mathbf{true}) : \mathbf{int} \times \mathbf{bool}}{\vdash \mathbf{let} f = \mathbf{fun} x \rightarrow x \mathbf{in} (f \mathbf{1}, f \mathbf{true}) : \mathbf{int} \times \mathbf{bool}}}$$

Primitives

Are now handled satisfactorily :

$$fst : \forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \alpha$$

$$snd : \forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \beta$$

$$opif : \forall\alpha.\mathbf{bool} \times \alpha \times \alpha \rightarrow \alpha$$

$$opfix : \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha$$

Exercise

Type!

$$\Gamma \vdash \text{fun } x \rightarrow x x : (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \alpha)$$

Remark

The condition $\alpha \notin \text{FV}(\Gamma)$ in the rule :

$$\frac{\Gamma \vdash e : \tau \quad \alpha \notin \text{FV}(\Gamma)}{\Gamma \vdash e : \forall \alpha. \tau}$$

is crucial

without it, we would have :

$$\frac{\frac{\Gamma + x : \alpha \vdash x : \alpha}{\Gamma + x : \alpha \vdash x : \forall \alpha. \alpha}}{\Gamma \vdash \mathbf{fun} \ x \rightarrow x : \alpha \rightarrow \forall \alpha. \alpha}$$

And the following program would be accepted :

`(fun x → x) 1 2`

Bad news !

For non annotated terms, the two problems :

- *inference* : Given e , does-there exist τ such that $\vdash e : \tau$?
- *verification* : Given e and τ , do we have $\vdash e : \tau$?

are **undecidable**.

J. B. Wells. *Typability and type checking in the second-order lambda-calculus are equivalent and undecidable*, 1994.

► The *Hindley-Milner* (Ocaml, ...) impose a syntactic restriction for program to be well-typed. (see ref in the course webpage).

Ocaml notations for parametric types

```
# fst;;
```

```
- : 'a * 'b -> 'a = <fun>
```

$$\forall \alpha. \forall \beta. \alpha \times \beta \rightarrow \alpha$$

```
# List.fold_left;;
```

```
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

$$\forall \alpha. \forall \beta. (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta \text{ list} \rightarrow \alpha$$