Compilation and Program Analysis (#4) : Types, Typing

Laure Gonnord

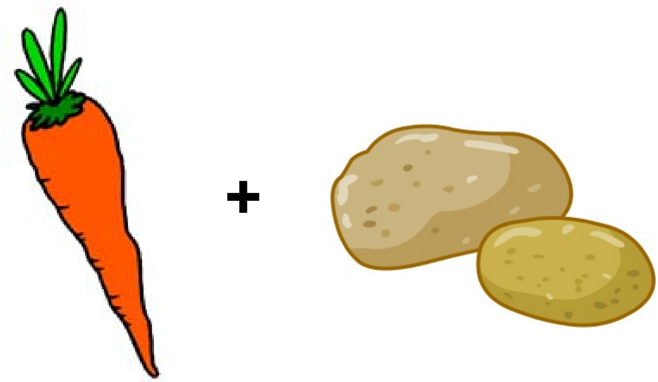
http://laure.gonnord.org/pro/teaching/capM1.html

Laure.Gonnord@ens-lyon.fr

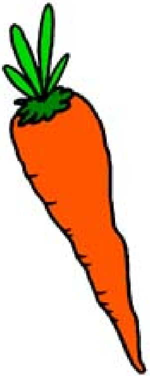
Master 1, ENS de Lyon

oct 2016

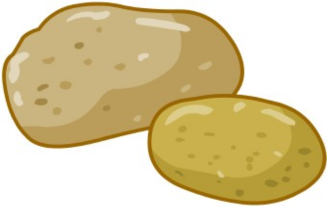
Typing



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 alaysis 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 arror in the eveluation 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 anotated by a type

$$\texttt{fun}\ (x : \texttt{int}) \rightarrow \texttt{let}\ (y : \texttt{int}) = (+ :)(((x : \texttt{int}), (1 : \texttt{int})) : \texttt{int} \times \texttt{int})\ \texttt{in}$$

  easy to verify, but tedious for the programmer

- Annotate only variable declarations (Pascal, C, Java, ...)

$$\texttt{fun}\ (x : \texttt{int}) \rightarrow \texttt{let}\ (y : \texttt{int}) = +(x, 1)\ \texttt{in}\ y$$

- Only annotate function parameters

$$\texttt{fun}\ (x : \texttt{int}) \rightarrow \texttt{let}\ y = +(x, 1)\ \texttt{in}\ y$$

- 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.

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.

Mini-While Syntax

Expressions :

e	$::=$	c	*constant*
	$\mid$	x	*variable*
	$\mid$	$e + e$	*addition*
	$\mid$	$e \times e$	*multiplication*
	$\mid$	$\ldots$	

Mini-while :

$S(Smt)$	$::=$	$x := expr$	assign
	$\mid$	$skip$	do nothing
	$\mid$	$S_1; S_2$	sequence
	$\mid$	$\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$	test
	$\mid$	$\texttt{while } b \texttt{ do } S \texttt{ done}$	loop

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

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\text{(or bool, ...)}$$

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

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 !

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

MyMuTypingVisitor.py

```
def visitAdditiveExpr(self, ctx):
    lvaltype = self.visit(ctx.expr(0))
    rvaltype = self.visit(ctx.expr(1))

    op = self.visit(ctx.oplus())
    if lvaltype == rvaltype:
        return lvaltype
    elif {lvaltype, rvaltype} == {BaseType.Integer, BaseType
        .Float}:
        return BaseType.Float
    elif op == u'+' and any(vt == BaseType.String for vt in
        (rvaltype, lvaltype)):
        return BaseType.String
    else:
        raise SyntaxError("Invalid type for additive operand
            ")
```

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

```
    def __init__(self):
        super(Expression, self).__init__()

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

class AddOp(BinOp):
```

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.pmop()) == '+'):  #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)

Later

In the course dedicated to functions in mini-while.

Mini-ml typing

- monomorphic typing
- polymorphism, type inference

Mini-ML, syntax recall

e	$::=$	x	ident
	$\mid$	c	constant (1, 2, ..., *true*, ...)
	$\mid$	op	primitive ($+$, $\times$, *fst*, ...)
	$\mid$	$\texttt{fun}\ x \rightarrow e$	function
	$\mid$	$e\ e$	application
	$\mid$	(e, e)	pair
	$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$	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 :

$$\begin{array}{lcll} \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \textit{base type} \\ & \mid & \tau \rightarrow \tau & \textit{fonction type} \\ & \mid & \tau \times \tau & \textit{type} \end{array}$$

Typing rules for mini-ml

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \mathtt{int}}\cdots \qquad \frac{}{\Gamma \vdash + : \mathtt{int} \times \mathtt{int} \to \mathtt{int}}\cdots$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

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

Example

$$
\dfrac{
\dfrac{
\dfrac{\vdots \qquad \dfrac{\vdots}{x : \mathtt{int} \vdash (x, 1) : \mathtt{int} \times \mathtt{int}}}{x : \mathtt{int} \vdash +(x, 1) : \mathtt{int}}
}{\emptyset \vdash \mathtt{fun}\ x \to +(x, 1) : \mathtt{int} \to \mathtt{int}}
\qquad
\dfrac{\dfrac{}{\ldots \vdash f : \mathtt{int} \to \mathtt{int}} \qquad \dfrac{}{\ldots \vdash 2 : \mathtt{int}}}{f : \mathtt{int} \to \mathtt{int} \vdash f\ 2 : \mathtt{int}}
}{\emptyset \vdash \mathtt{let}\ f = \mathtt{fun}\ x \to +(x, 1)\ \mathtt{in}\ f\ 2 : \mathtt{int}}
$$

Non typable expressions

We cannot type `1 2` :

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

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

$$\frac{\dfrac{\Gamma + x : \tau_1 \vdash x : \tau_3 \rightarrow \tau_2 \qquad \Gamma + x : \tau_1 \vdash x : \tau_3}{\Gamma + x : \tau_1 \vdash x\ x : \tau_2}}{\Gamma \vdash \texttt{fun}\ x \rightarrow x\ x : \tau_1 \rightarrow \tau_2}$$

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

Several possible types

We can show :

$$\emptyset \vdash \texttt{fun}\ x \to x : \texttt{int} \to \texttt{int}$$

but also :

$$\emptyset \vdash \texttt{fun}\ x \to x : \texttt{bool} \to \texttt{bool}$$

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

Several possible types 2/2

The term $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$ is not typable because there is no τ such that :

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

nevertheless :

$$((\texttt{fun}\ x \to x)\ (\texttt{fun}\ x \to x))\ \ 42$$

is typable (exercise !)

Primitives : pb

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

$$\texttt{int} \times \texttt{int} \to \texttt{int}$$
$$\texttt{int} \times \texttt{bool} \to \texttt{int}$$
$$\texttt{bool} \times \texttt{int} \to \texttt{bool}$$
$$(\texttt{int} \to \texttt{int}) \times \texttt{int} \to \texttt{int} \to \texttt{int}$$

etc.

Typing functions of mini-ML

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

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 environnement (Γ) is a persistant 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* $\emptyset \vdash e' : \tau$*.*

Polymorphism for mini-ML

Give a unique type to $\texttt{fun}\ x \rightarrow x$ in

$$\texttt{let}\ f = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ \ldots$$

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 :

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \textit{base types} \\
 & \mid & \tau \rightarrow \tau & \textit{function type} \\
 & \mid & \tau \times \tau & \textit{pair} \\
 & \mid & \alpha & \textit{variable type} \\
 & \mid & \forall \alpha . \tau & \textit{polymorphic type}
\end{array}
$$

F-system

Same rules plus :

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha . \tau}$$

and

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

$\mathcal{L}(\Gamma)$ == free variables of Γ

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

Example

$$\dfrac{\dfrac{\dfrac{x : \alpha \vdash x : \alpha}{\vdash \mathtt{fun}\ x \to x : \alpha \to \alpha}}{\vdash \mathtt{fun}\ x \to x : \forall \alpha.\alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{\dots \vdash f : \forall\alpha.\alpha \to \alpha}{\dots \vdash f : \mathtt{int} \to \mathtt{int}} \quad \vdots}{\dots \vdash f\ 1 : \mathtt{int}} \qquad \dfrac{\vdots}{\dots \vdash f\ \mathtt{true} : \mathtt{bool}}}{f : \forall\alpha.\alpha \to \alpha \vdash (f\ 1, f\ \mathtt{true}) : \mathtt{int} \times \mathtt{bool}}}{\vdash \mathtt{let}\ f = \mathtt{fun}\ x \to x\ \mathtt{in}\ (f\ 1, f\ \mathtt{true}) : \mathtt{int} \times \mathtt{bool}}$$

Primitives

Are now handled satisfactorily :

$$\begin{aligned} fst : \quad & \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha \\ snd : \quad & \forall\alpha.\forall\beta.\alpha \times \beta \to \beta \\ opif : \quad & \forall\alpha.\mathtt{bool} \times \alpha \times \alpha \to \alpha \\ opfix : \quad & \forall\alpha.(\alpha \to \alpha) \to \alpha \end{aligned}$$

Exercise

Type !

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

Remark

The condition $\alpha \notin \mathcal{L}(\Gamma)$ in the rule :

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha . \tau}$$

is crucial

without it, we would have :

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

And the following program would be accepted :

$$(\texttt{fun}\ x \rightarrow x)\ 1\ 2$$

Bad news !

For non annoted 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 syntaxic restriction for program to be well-typed. (see ref in the course webpage).

Ocaml notations for parametric types

```
# fst;;
```

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

$$\forall \alpha . \forall \beta . \alpha \times \beta \to \alpha$$

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

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

$$\forall \alpha . \forall \beta . (\alpha \to \beta \to \alpha) \to \alpha \to \beta \; list \to \alpha$$