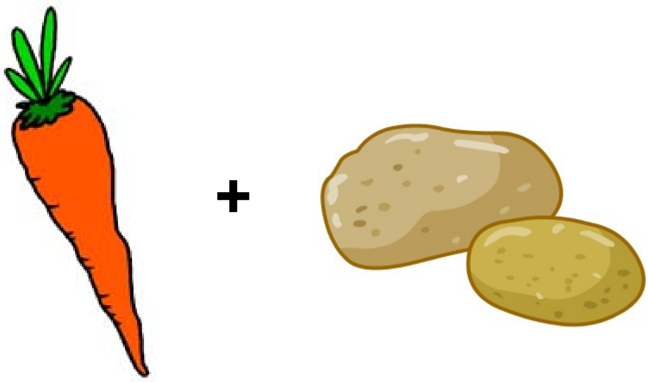
Types, Typing

MIF08

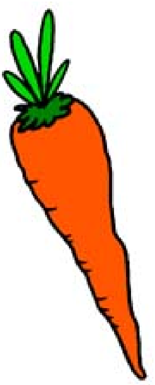
Laure Gonnord
`Laure.Gonnord@univ-lyon1.fr`

oct 2016

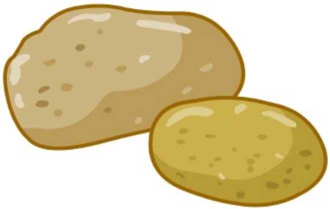
Typing



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 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 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.

Outline

Mini-While Syntax

Expressions:

$$\begin{array}{rcll} e & ::= & c & \textit{constant} \\ & \mid & x & \textit{variable} \\ & \mid & e + e & \textit{addition} \\ & \mid & e \times e & \textit{multiplication} \\ & \mid & \ldots & \end{array}$$

Mini-while:

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

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

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

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

Hybrid expressions

What if we have `1.2 + 42` ?

- reject?
- compute a float!

▶ This is **type coercion**.

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

Idea: the type is void otherwise “typing error”

$$\frac{\Gamma \vdash e : t \quad \Gamma(x) : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x := e : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} : \texttt{void}}$$

Outline

Principle of type checking

- 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
                ")
```

Typing is more than type checking

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

And we want **informative errors**:

```
Type error at line 42
```

is not sufficient!

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)
```