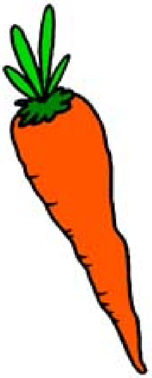
Compilation and Program Analysis (#4) : Types, and Typing MiniWhile

Laure Gonnord & Matthieu Moy & other
https://compil-lyon.gitlabpages.inria.fr/

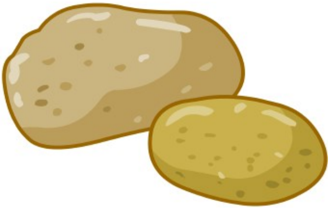
Master 1, ENS de Lyon et Dpt Info, Lyon1

2021-2022

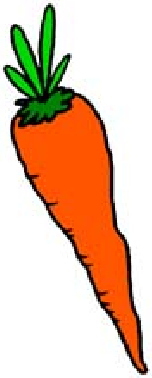
Typing



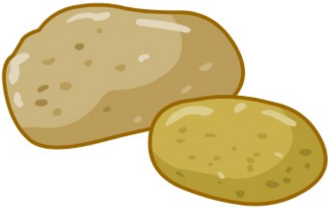
+



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

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

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

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, …)

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

- Only annotate function parameters

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

- 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* $e \to e'$*.*

Lemme (preservation)

If $\emptyset \vdash e : \tau$ *and* $e \to e'$ *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. Typing ML - ENSL Only

3. Imperative languages (C, Mini-While)
   - Simple Type Checking for mini-while
   - A bit of implementation (for expr)
   - Safety ENS Only

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 rules for expr

Here types are basic types: Int|Bool

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \texttt{int}}$$ (or $\mathrm{tt}\colon\ \mathrm{bool}, \ldots$)

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

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

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

on board!

Typing While : recap

$$\frac{c \in \mathbb{Z}}{\Gamma \vdash c : \texttt{int}} \qquad \frac{\Gamma(x) = t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x : t}$$

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

$$\frac{\Gamma \vdash S_1 : \texttt{void} \quad \Gamma \vdash S_2 : \texttt{void}}{\Gamma \vdash S_1 ; S_2 : \texttt{void}} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x = e : \texttt{void}}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} : \texttt{void}}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S_1 : \texttt{void} \quad \Gamma \vdash S_2 : \texttt{void}}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \texttt{void}}$$

Typing: an example

Considering $\Gamma = \{x_1 \mapsto int\}$, prove that the given sequence of instructions is well typed:

```
x1 = 3 ;
x1 = x1+9 ;
```

on board!

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.

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 ENSL Only

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

$$int <: num$$

- Need additional rule to use sub-typing:

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

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

$$\frac{e : Array[\tau] \qquad \tau <: \tau'}{e : Array[\tau']}$$

How to define subtyping for functions?

Note: subtyping is heavily used in OOP

1. Generalities about typing

2. Typing ML - ENSL Only

3. Imperative languages (C, Mini-While)
   - Simple Type Checking for mini-while
   - A bit of implementation (for expr)
   - Safety ENS Only

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

1. Generalities about typing

2. Typing ML - ENSL Only

3. Imperative languages (C, Mini-While)
   - Simple Type Checking for mini-while
   - A bit of implementation (for expr)
   - Safety ENS Only

Summary

1. Generalities about typing
2. Typing ML - ENSL Only
3. Imperative languages (C, Mini-While)
   - Simple Type Checking for mini-while
   - A bit of implementation (for expr)
   - Safety ENS Only