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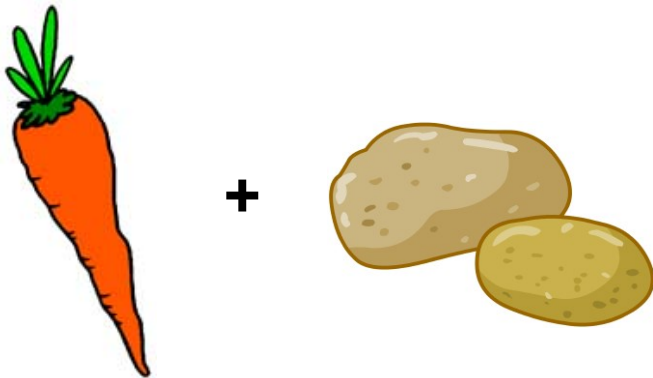
Ludovic Henrio

Master 1, ENS de Lyon et Dpt Info, Lyon1

2025-2026

Lyon 1

ENS DE LYON

Typing

+

Typing and Behaviour

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

$$"5" + 37$$

- a compilation error? (OCaml)

- an execution error? (Python)

- the int 42? (Visual Basic, PHP)

- the string "537"? (Java, JavaScript)

- anything else?

Typing and Behaviour

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

$$"5" + 37$$

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

We have two camps here: OCaml and Python somehow
complain, the others... find a way.

Typing and Behaviour

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

$$"5" + 37$$

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

We have two camps here: OCaml and Python somehow complain, the others... find a way. ... but in fact the moment when the language complain/adapt is more important! is often

So, what is it about?

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*
>
> (Pierce, *Types and Programming Languages*, 2002)

Let's type then

The broader question becomes: when should the program

```
e1 + e2
```

be legal? And what is its semantics?

▶ Typing: an <u>analysis</u> that associates a <u>type</u> to each term of the language, and rejects programs that are considered incoherent.

When?

- Dynamic typing (during execution): Lisp, PHP, Python, JavaScript
- Static typing (at compile time, after lexing+parsing): C, Java, OCaml

When?

- Dynamic typing (during execution): Lisp, PHP, Python, JavaScript

- Static typing (at compile time, after lexing+parsing): C, Java, OCaml

- Hybrid: allow typing annotations on dynamically typed languages (Python with mypy or Pyright, JavaScript with TypeScript, etc.). See also: Gradual Typing.

▶ This course: **static typing**.

Slogan

well typed programs cannot go wrong

Milner, "A Theory of Type Polymorphism in Programming", 1978

Slogan

well typed programs cannot go wrong

Milner, "A Theory of Type Polymorphism in Programming", 1978

(For some definition of "well-typed" and "go wrong"...)

What are type systems good for?

- Detecting <u>some</u> programming errors
- Abstraction: modules, interfaces, parametricity...
- Documentation
- Safety
- Efficient compilation!

Typing objectives

- Should be **decidable**.

- It should reject programs like (1 2) in OCaml, or
  1.0+"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**)

Rough principle

We type recursively the
sub-expressions.
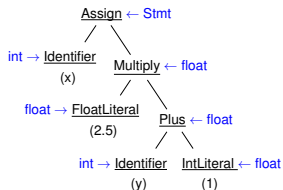
What does the programmer write?

- The type of all sub-expressions (like above) easy to verify, but
  tedious for the programmer
- Annotate only variable declarations (Pascal, C, Java, ...)
  ```
  {int x, y; x = 2.5 * (y + 1);}
  ```
- Only annotate function parameters (Scala)
  ```
  def foo(y : Int) { var x = 2.5 * (y + 1) }
  ```
- Annotate nothing, rely on inference : OCaml, Haskell, ...
  ```
  # let foo y = 2 * (y + 1);;
  val foo : int -> int = <fun>
  ```

Typing algorithm

Note that we distinguish:

- the typing algorithm, an effective process that takes as output a possibly partially annotated program and either reject it, or outputs its type.

- the typing system, a specification of the well-typed programs

The former should be well-behaved w.r.t. the latter, i.e.,:

- <u>Correctness</u>: accepts only well-typed programs.

- <u>Completeness</u>: accepts any well-typed program.

In some cases, we can also aim at:

- <u>principality</u> : The most general type is computed.

Typing judgement

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

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

stating that "in the environment Γ, expression e has type τ"

▶ Γ associates a type to the the variables that may appear in the expressions.

```
{
    int x = 42, y;
    float z;
}
```
$\rightsquigarrow$
$$\{ \\ x \rightarrow int, \\ y \rightarrow int, \\ z \rightarrow float \\ \}$$

Type safety, i.e., well typed programs cannot go wrong

In general a type safety theorem looks like this:

Theorem (Safety)

If $\emptyset \vdash e : \tau$, then the reduction of e is infinite, or it terminates in a valid final configuration.

The notions of valid final configuration is vague. For expressions, it would be a value. In our mini-while it is a final configuration σ.

What the theorem really captures is: there will be no runtime error. Again the runtime errors captured depend on the semantics and the type system. —a notion, once again, that depends on the semantics and the type system,.

Type Safety: proof methodology

The standard proof methodology is based on two lemmas:

Lemme (Progress)

If $\emptyset \vdash e : \tau$, then either e is final or there exists e' such that $e \to e'$.

Lemme (Preservation)

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

Together, these lemmas imply type safety.

Small-step or big-step semantics are proven (more or less) the same way.

Mini-While Syntax

Expressions (with boolean expression to make typing interesting):

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
  & | & x & \textit{variable} \\
  & | & e + e & \textit{addition} \\
  & | & e < e & \text{boolean expressions} \\
  & | & ... &
\end{array}
$$

Mini-while:

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

Typing rules for expr

Only two ground types: int | bool

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

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

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \wedge e_2 : \texttt{bool}} \qquad\qquad \dots$$

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 <u>is well-typed</u> (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 + find a program that is not well typed

Problem: how to define Γ in mini-while?

One possible solution(used in the following): <u>programs declare variables</u>.

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

We can then simply define $\Gamma_D \triangleq \{x \mapsto \tau \text{ s.t. } x : \tau \in D\}$.

Γ_D is <u>undefined</u> if $x : \tau \in D$ and $x : \tau' \in D$ with $\tau \neq \tau'$

And type programs as:

$$
\frac{\Gamma_D \vdash S}{\emptyset \vdash D; S}
$$

Typing judgement for runtime configuration

The semantics of Mini-While does not operate on programs, but on configurations, i.e., programs paired with a store. This is the case for the initial state, and the intermediate states of the small-step semantics.

In order to reason on types at runtime, we extend the typing judgement to these runtime configurations:

Definition (Configuration typing)

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

The second part somehow says, the typing environment is a correct abstraction of the runtime store.

Note : $\sigma(x)$ is a value: no Γ is needed to type it.

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 ...

Coercions

Assuming we extend our language with floats, what should we do with `1.2 + 42` ?

- reject?
- compute a float!

▶ This is a case of **type coercion**.

▶ It requires a very local form of <u>type inference</u>.

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.

Subtyping: heavily used in OOP notably

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

$$int <: num$$

- The subtyping relation can be used to weaken typing:

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

- The subtyping relation can be tricky:

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

$$\frac{\tau <: \tau'}{\tau' \to unit <: \tau \to unit} \quad \texttt{Contravariance}$$

The case of expressions

Typical case of correctness for an evaluation in one-step (big-step).

Safe typing for $Val(e, \sigma)$ is expressed as follows.

Theorem (Safety)

Suppose $\forall x \in vars(e).\ \emptyset \vdash \sigma(x) : \tau \iff \Gamma(x) = \tau$

Then $\Gamma \vdash e : \tau \implies \emptyset \vdash Val(e, \sigma) : \tau$

Prove it!

Typing judgment for statements: reminder

$$\frac{\Gamma_D \vdash S}{\emptyset \vdash D; S}$$

With $\Gamma_D \triangleq \{x \mapsto \tau \text{ s.t. } x : \tau \in D\}$ (undefined if multiple defs).

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \qquad \Gamma \vdash x : \Gamma(x) \qquad \frac{c \in \mathbf{Z}}{c : int}$$

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

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

Typing judgment for configurations

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

Safety = well typed programs cannot go wrong

In case of a <u>small-step semantics</u> the proof that "well typed programs cannot 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 $(S, \sigma) \rightarrow \sigma'$.

... and remain well-typed

Lemma (preservation)

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

Note that Γ never changes (defined by declarations)

Proofs! (recall the property for expression evaluation)

Initial Configuration

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

BUT we need to initialize the store in a way that is compatible with the typing environment: **Γ and $\emptyset$ do not agree on the typing**

Three basic solutions:

- Enforce declarations to come with initialization or

- Define a default value for each type.

- consider that $\sigma(x)$ agrees with anything if $x \notin Dom(\sigma)$

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 (and columns).

- Output: Trees are decorated by types in addition.

In practice for mini-C (lab sessions)

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

We can associate type to variables, directly from parsing :

Gamma is constructed with lexing information or parsing (variable declaration with types).

Conclusion 1/2

We have seen:

- The principle of static typing

- A type system for mini-while

- Type safety and how to prove it for mini while

Conclusion 2/2

Further discussions not covered here:

- Typing functions (later in the course)

- More complex (i.e. real life) type system: sub-typing, objects, polymorphisms, modules, type classes...

- There exist very rich type systems , e.g. behavioural types, session types, linear types, ownership types, liquid types ...
  It is an old but still active and exciting area of research!

Undefined behaviors as a valid feature

Compiler must be able to inherit from invariants, without worsening the runtime!
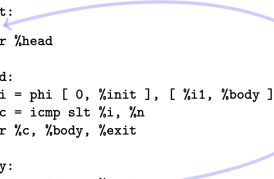
Quickly a need: delayed undefined behaviors!

```
for (int i = 0; i < n; ++i) {
  a[i] = x + 1;
}
```
———————————————————————————

```
init:
  br %head

head:
  %i = phi [ 0, %init ], [ %i1, %body ]
  %c = icmp slt %i, %n
  br %c, %body, %exit

body:
  %x1 = add nsw %x, 1
  %ptr = getelementptr %a, %i
  store %x1, %ptr
  %i1 = add nsw %i, 1
  br %head
```
———————————————————————————

Figure 1. C code and its corresponding LLVM IR. We want to hoist the invariant addition out of the loop. The nsw attribute means the add is undefined for signed overflow.

But what about unsafe features?

Static typing are great! My favorite language is strongly, statically typed, and of course type safe!

But what about unsafe features?

Static typing are great! My favorite language is strongly, statically typed, and of course type safe!

- OCaml: Obj.magic
- Haskell: unsafePerformIO, etc...
- Rust: unsafe blocks

But what about unsafe features?

Static typing are great! My favorite language is strongly, statically typed, and of course type safe!

- OCaml: Obj.magic
- Haskell: unsafePerformIO, etc. . .
- Rust: unsafe blocks

But I only used these features carefully in some well crafted library code! So surely my language is still type safe?
Well yes, but syntactic type safety is of no help to make sure of that.

$\rightarrow$ semantic typing