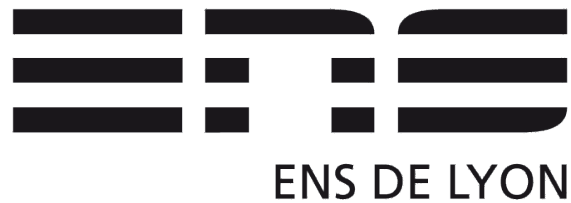
Compilation and Program Analysis (#3) : Types, and Typing MiniWhile

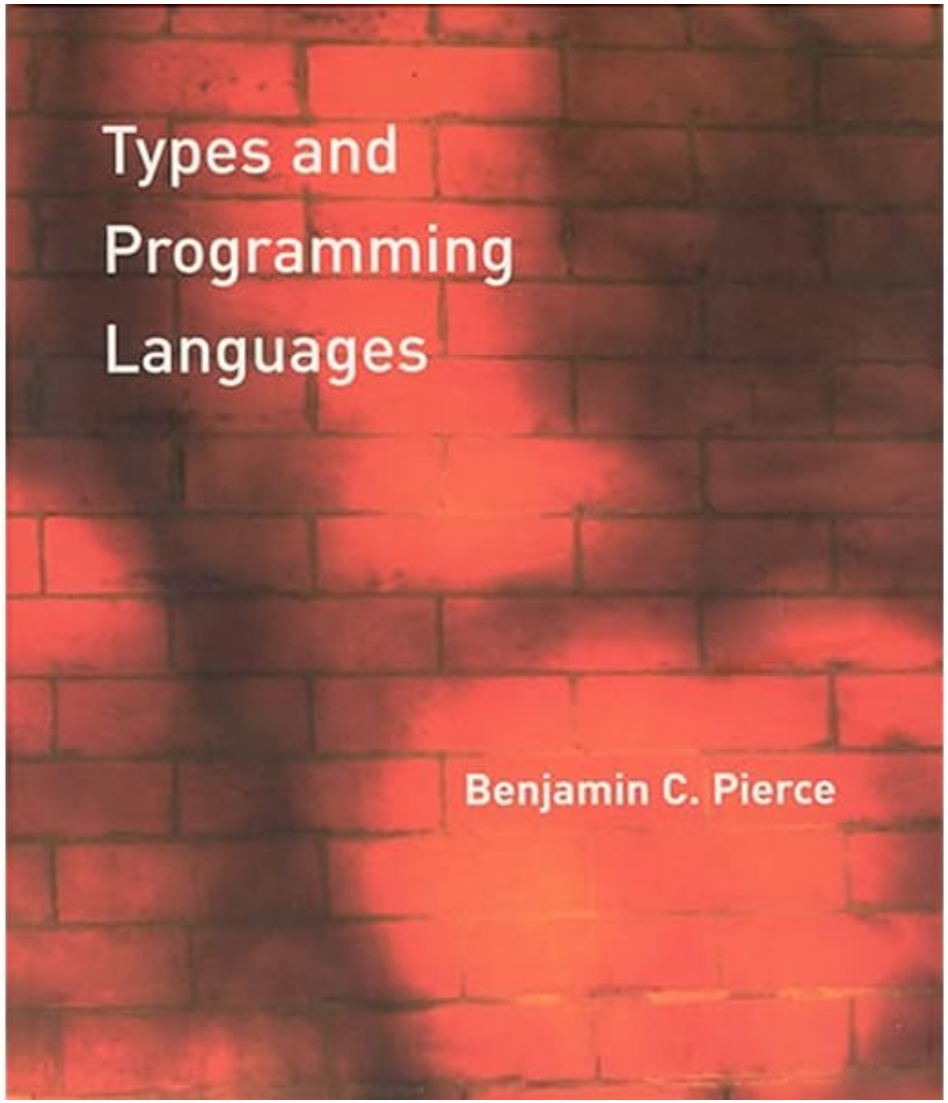
Yannick Zakowski

Master 1, ENS de Lyon et Dpt Info, Lyon1

2024-2025



Book of the week



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)

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)

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)

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)

Couteau à viande, couteau à poisson

How should the following program behave?

```
"5" + 37
```

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

Couteau à viande, couteau à poisson

How should the following program behave?

```
"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.

Let's type then

The broader question becomes: when should the program

```
e1 + e2
```

be legal? And what of its semantics can and must I know to understand what contexts can legally use this program?

- The art of typing: an analysis that associates a type to each term of the language, and rejects undesired programs.

Let's type then

The broader question becomes: when should the program

```
e1 + e2
```

be legal? And what of its semantics can and must I know to understand what contexts can legally use this program?

- The art of typing: an analysis that associates a type to each term of the language, and rejects undesired programs.

Programs are the terms generated by the BNF.

Let's type then

The broader question becomes: when should the program

```
e1 + e2
```

be legal? And what of its semantics can and must I know to understand what contexts can legally use this program?

- The art of typing: an analysis that associates a type to each term of the language, and rejects undesired programs.

  Programs are the well typed terms generated by the BNF.

To type, but when?

Typing error

```
print("Hello")
x = 1.0 + "you can't add a string to a float"
```

$\rightsquigarrow$ Will the program get to scream "Hello" before erroring in agony?

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**.

What are type systems good for?

- Detecting some programming errors
- Abstraction: modules, interfaces, parametricity...
- Documentation
- Safety, but not necessarily (ML, Java, Lips: yes; C, C++: no)
- Efficient compilation!

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”...)

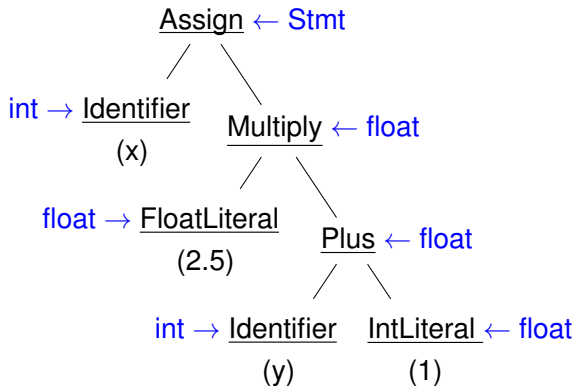
Typing objectives

- 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.,:

- Correctness: accepts only well-typed programs.
- Completeness: accepts any well-typed program.

In presence of polymorphism, we are also often concerns with:

- principality : 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 variables in scope.

```
{
    int x = 42, y;
    float z;
}
```

$\rightsquigarrow$

$$\begin{array}{l} \{ \\ x \rightarrow int, \\ y \rightarrow int, \\ z \rightarrow float \\ \} \end{array}$$

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

In general a type safety theorem has the following flavor:

Theorem (Safety)

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

Of course, the notions of valid final configuration are language-dependent. For a calculus, it would be a value. In our mini-while it is a final configuration $(\texttt{skip}, \sigma)$.

What the theorem really captures is: there will be no runtime error—a notion, once again, that depends on your semantics.

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 implies immediately type safety.

A relatively recent recipe

A Syntactic Approach to Type Soundness

Andrew K. Wright* Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892

June 18, 1992

Rice Technical Report TR91-160
To appear in: *Information and Computation*

Abstract

We present a new approach to proving type soundness for Hindley/Milner-style polymorphic type systems. The keys to our approach are (1) an adaptation of subject reduction theorems from combinatory logic to programming languages, and (2) the use of rewriting techniques for the specification of the language semantics. The approach easily extends from polymorphic functional languages to imperative languages that provide references, exceptions, continuations, and similar features. We illustrate the technique with a type soundness theorem for the core of Standard ML, which includes the first type soundness proof for polymorphic exceptions and continuations.

Typing acts upon the abstract syntax

Expressions:

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

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`

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

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

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \texttt{bool}} \qquad \ldots$$

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

A statement S is well-typed (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 ;
```

Problem: how to define Γ in mini-while?

Possible solution: programs declare variables.

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

We can then simply define $\Gamma_D \triangleq x \mapsto \tau$ `iff` $x : \tau \in D$.

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.[1]
In order to reason on types at runtime, for instance for preservation, we hence need to extend the typing judgement to these runtime configurations:

Definition (Configuration typing)

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

That is to say, the typing environment is a correct abstraction of the runtime store.

[1] Remark that semantics based on substitution do not have need for this notion

2 Imperative languages (C, Mini-While)

- Simple Type Checking for Mini-While
- Other typing features

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 type inference.

Structural vs nominative type systems

In presence of user defined datatype, how should we compare for the compatibility of types?

```
typedef struct {
  int data[100];
  int count;
  } Stack;
```

```
typedef struct {
  int data[100];
  int count;
  } Set;
```

- Structural discipline: Stack and Set are compatible.
- Nominative discipline: they are not.

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

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 Mini While: reminder

$$\emptyset \vdash (D, P) : \tau \quad \texttt{if} \quad \Gamma_D \vdash P : \tau$$

$$\frac{D \to_d \Gamma \quad \Gamma \vdash S}{\emptyset \vdash D; S} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \Gamma \vdash x : \Gamma(x)$$

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

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

Typing configurations:

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

Safety = well typed programs do not go wrong

In case of a small-step semantics the proof that "well typed programs do not go wrong" relies on two lemmas:

Well-type programs run without error

Lemma (Progress)

If $\Gamma \vdash (S, \sigma)$*, then there exists* S', σ' *such that* $(S, \sigma) \to (S', \sigma')$
OR $S = \texttt{skip}$*.*

... and remain well-typed

Lemma (Preservation, a.k.a. subject reduction)

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

Note that Γ never changes (defined by declarations)

Proofs! (recall the property for expression evaluation)

Initial Configuration

Remark: we need to initialize the store in a way that is compatible with the typing environment.

Two basic solutions:

- Enforce declarations to come with initialization or
- Define a default value for each type.

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. session types, linear types, ownership types, liquid types ...
  It is an old but still very active and exciting area of research!

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.

Semantic Typing

How did we do before 1994?

- Syntactic typing $\Gamma \vdash t : \tau$
- Semantic $\Gamma \vDash t : \tau$

We capture terms that are syntactically ill-formed, but behave safely: "t behaves safely when used at type τ".

Semantic Typing

$$\Gamma \vDash t : \tau$$

- Adequacy: if $\emptyset \vDash t : \tau$ then t is safe
- Compatibility: $\vDash$ is compatible with the syntactic typing rules

We capture terms that are syntactically ill-formed, but behave safely: "t behaves safely when used at type τ".

In particular, $\Gamma \vdash t : \tau \Rightarrow \Gamma \vDash t : \tau$

Application: the RustBelt project

See Derek Dreyer's Milner Award Lecture