SW7 Programming Paradigms

Static Type Systems

block 1: typing of a functional language

Léon Gondelman

AALBORG UNIVERSITY | COPENHAGEN | 2026

Mars Climate Orbiter, 1999

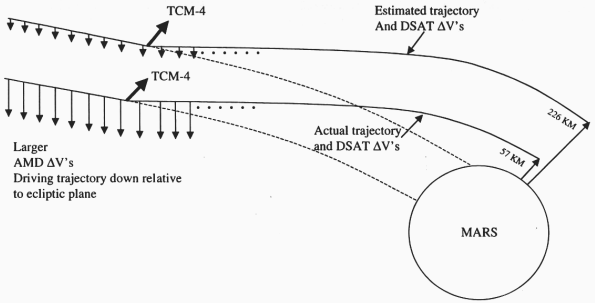


figure 4 of the investigation board's report (NASA, 1999)

Lockheed Martin's program writes impulses in **pound-force seconds**

JPL's program reads them as **newton-seconds**

a factor 4.45: 57 km above Mars instead of 226 km, and the spacecraft is **lost**

what would have caught it?

one type for each unit: the mix becomes a **type error**, reported before the run

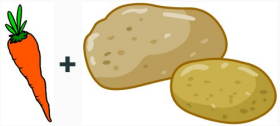
type checking

an operation applied to a value of the wrong type:

```
"5" + 37
```

do we get

- a compile-time error? (OCaml, Rust, Go)
- a runtime error? (Python, Julia)
- the number 42? (PHP; Visual Basic with Option Strict Off)
- the string `"537"`? (Java, Scala, Kotlin)
- an out-of-bounds pointer? (C, C++)
- something else?



and what does this one give?

```
37 / "5"
```

types, intuitively

a **type** names a collection of values, together with the operations defined on them

type	values	operations
`int`	$\ldots, -1, 0, 1, 2, \ldots$	$+$, $\times$, $<$
`bool`	`true`, `false`	conditional, negation
`string`	`""`, `"a"`, `"ab"`, $\ldots$	concatenation, length
`int` $\to$ `bool`	the functions from integers to Booleans	application

a **type error** is an operation applied to a value on which it is not defined: `length(42)`
`1 + length 42("abc")`

types: a second reading

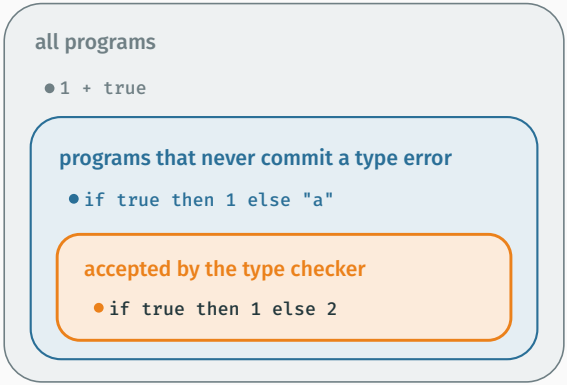
the intuitive reading is an approximation: a type is assigned to an **expression**, in the text of the program, before any value exists

- `if true then 1 else "a"` evaluates to the integer 1, and OCaml gives it no type: typing judges the text, not one run
- for a function, the type `int` $\rightarrow$ `int` is a claim about **every** call; no single value shows it
- a stack and the list that implements it hold the same values, and are two types: a type also **hides** a representation

"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, 2002

"Type structure is a syntactic discipline for enforcing levels of abstraction." Reynolds, 1983

static typing: a sound approximation



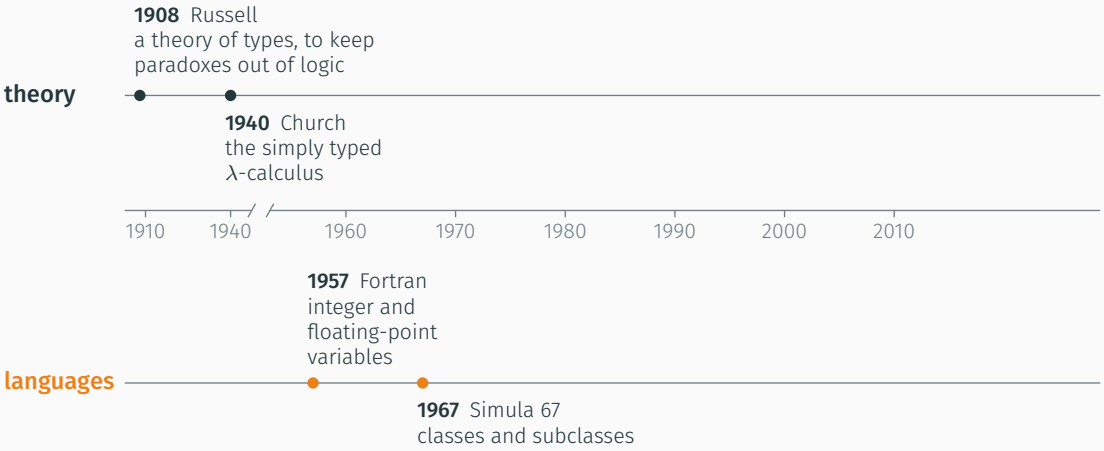
can a checker accept exactly the blue set?

no: whether a program ever commits a type error is **undecidable**

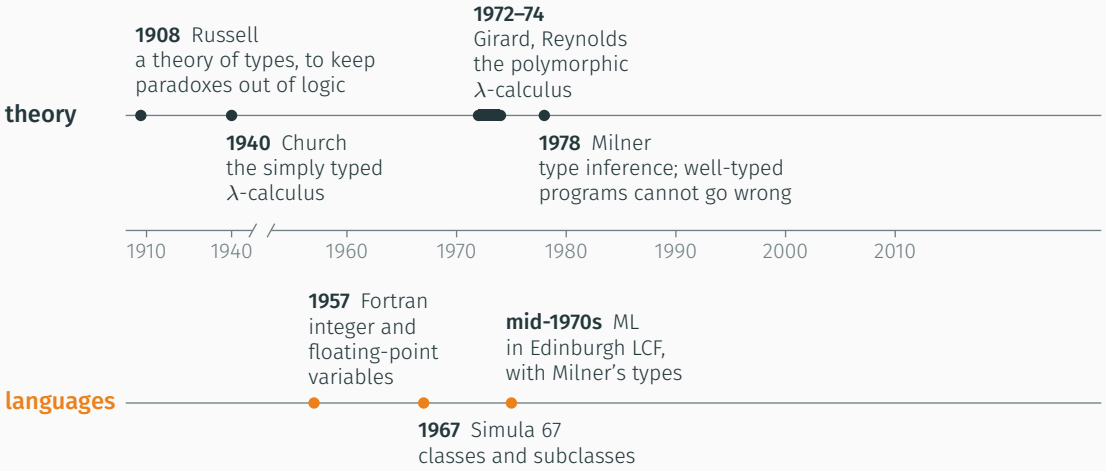
a type checker is

- **decidable**: it always answers
- **sound**: what it accepts is in the blue set (when the type system is safe)
- hence **incomplete**: it rejects some programs of the blue set

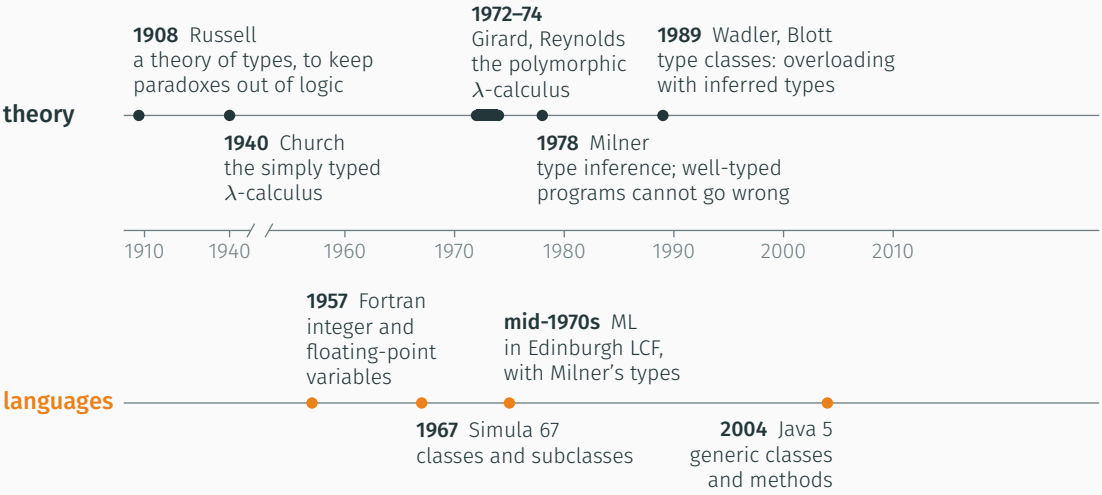
types in programming languages: a timeline



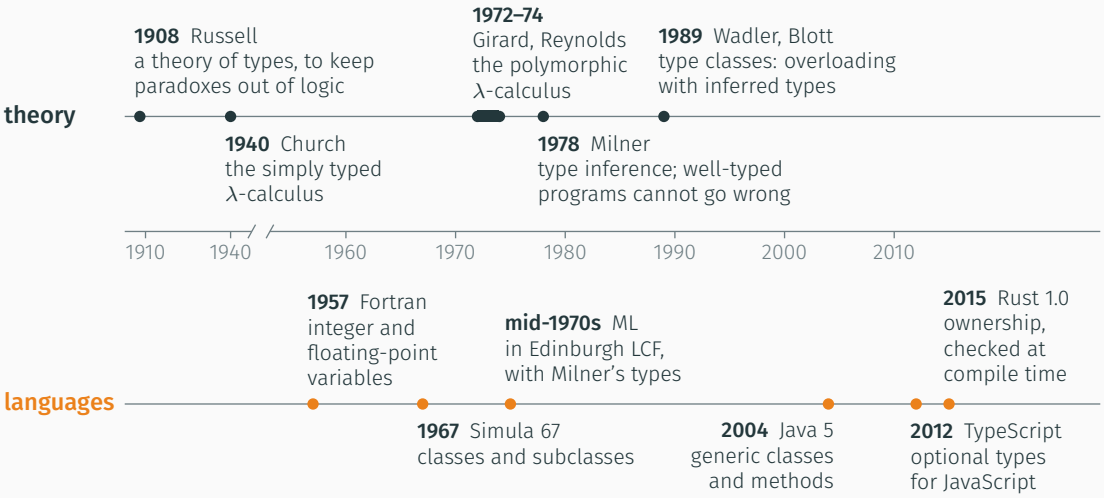
types in programming languages: a timeline



types in programming languages: a timeline



types in programming languages: a timeline



types as proofs: some dates

theory

1969 Howard, after Curry: a proposition is a type, a proof is a program



1988 Coquand, Huet: the calculus of constructions

a **kernel**, far smaller than the system, checks the proof, whoever or whatever wrote it: trust goes to the kernel, the logic and the statement



practice

1989 Coq, now Rocq: a proof assistant on that calculus

2005 the four-color theorem, checked in Coq (Gonthier, Werner)

2009 CompCert: a compiler for a large subset of C, proved correct in Coq (Leroy)

2013 Lean (de Moura)

2024 AlphaProof: proofs found by a machine, checked by Lean

what type systems check today

- **absence of null dereference**: in Kotlin and Swift, a value that may be absent has another type than a value that is present
- **memory safety without a garbage collector**: in safe Rust, a value has one owner; a use after the value was freed is a compile-time error
- **absence of data races**: in safe Rust, two threads that write the same data without synchronization do not compile (session 7)
- **theorems**: in Lean and Rocq, a proposition is a type, and a program of that type is its proof (session 1)

```
val name: String? = lookup(id)

name.length    // rejected
name?.length   // accepted
```

Kotlin: `name` may be absent; a direct access is rejected, an access that tests for absence is accepted

what type systems check today

- **absence of null dereference**: in Kotlin and Swift, a value that may be absent has another type than a value that is present
- **memory safety without a garbage collector**: in safe Rust, a value has one owner; a use after the value was freed is a compile-time error
- **absence of data races**: in safe Rust, two threads that write the same data without synchronization do not compile (session 7)
- **theorems**: in Lean and Rocq, a proposition is a type, and a program of that type is its proof (session 1)

```
let v = vec![1, 2, 3];
let first = &v[0];
drop(v);                // rejected
println!("{}", first);
```

Rust: `first` refers into `v`; freeing `v` while it is still referred to is rejected

what type systems check today

- **absence of null dereference**: in Kotlin and Swift, a value that may be absent has another type than a value that is present
- **memory safety without a garbage collector**: in safe Rust, a value has one owner; a use after the value was freed is a compile-time error
- **absence of data races**: in safe Rust, two threads that write the same data without synchronization do not compile (session 7)
- **theorems**: in Lean and Rocq, a proposition is a type, and a program of that type is its proof (session 1)

```
let mut n = 0;
thread::scope(|s| {
  s.spawn(|| n += 1);
  s.spawn(|| n += 1);  // rejected
});
```

Rust: two threads increment the same `n`; the second one is rejected

what type systems check today

- **absence of null dereference**: in Kotlin and Swift, a value that may be absent has another type than a value that is present
- **memory safety without a garbage collector**: in safe Rust, a value has one owner; a use after the value was freed is a compile-time error
- **absence of data races**: in safe Rust, two threads that write the same data without synchronization do not compile (session 7)
- **theorems**: in Lean and Rocq, a proposition is a type, and a program of that type is its proof (session 1)

```
theorem swap (p q : Prop)
    (h : p ∧ q) : q ∧ p :=
  ⟨h.2, h.1⟩
```

Lean 4: the statement "$p \wedge q$ implies $q \wedge p$" as a type, and its proof as a program that the checker accepts

typing

if we now add two arbitrary expressions

```
e1 + e2
```

how can we decide whether this is legal and which operation to perform?

the answer is **typing**, a *program analysis* that binds **types** to each sub-expression, to rule out inconsistent programs

when (to check/rule out inconsistencies)?

some languages are **dynamically typed**: types are bound to **values** and are used **at runtime**

examples: Lisp, PHP, Python, Julia

other languages are **statically typed**: types are bound to **expressions** and are used **at compile time**

examples: C, C++, Java, OCaml, Rust, Go

example

consider the following C and Python code snippets:

```c
int id(int num) {
  return num; }

void main(){
  printf("%d", id(42,42));}
```

```python
def id(num):
  return num

print(id(42,42))
```

the C code fails at the compile-time (compilation error)
error: too many arguments to function 'id'

the Python code compiles to the VM and fails at runtime (runtime error)
TypeError: id() takes 1 positional argument but 2 were given

static typing

slogan (Milner, 1978)

well-typed programs cannot “go wrong”

goals of typing

- type checking must be **decidable**
- type checking must reject programs whose evaluation would apply an operation to a value of the wrong type;
  this is **type safety**
- type checking must not reject too many non-absurd programs;
  the type system must be **expressive**

several solutions

1. any sub-expression is annotated with a type

```
int f(int x) { int y = ((x:int)+(1:int):int); ... }
```

   type checking is easy but this is unmanageable for the programmer

2. only annotate variable declarations (C, C++, Java, etc.)

```
int f(int x) { int y = x+1; return y; }
```

3. annotate parameters and results; local variables are inferred (`auto` in C++ 11, `var` in Java 10)

```
int f(int x) { var y = x+1; return y; }
```

4. no annotation at all ⇒ **type inference** (OCaml, Haskell, etc.)

```
fun x -> x+1
```

type safety, informally

a type system is **safe** when no program that it accepts applies, at run time, an operation to a value of the wrong type

safety is a property of the pair (type system, semantics): it is a **theorem**, and it has to be **proved**

does a program accepted by the compiler run without a type error?

in C, no: its type system is **not safe**; block 2 states the property formally

```c
#include <stdio.h>
union { long n; char *s; } u;

int main(void) {
  u.n = 42;
  printf("%s\n", u.s);
  return 0;
}
```

accepted by the C compiler without a warning; the run reads a text at address 42 and crashes

roadmap for today

- **block 1: typing of a functional language**
  - the simply typed λ-calculus: syntax, evaluation, typing rules
  - exercises by hand
  - mini-ML, and a type checker for it in OCaml (demo)
- **block 2: typing of an imperative language**
  - the WHILE language and its big-step semantics
  - typing of expressions and statements; type safety, stated
- **lab**: semantics and typing of `break` and `continue`

Simply Typed Lambda Calculus

1. recall of the λ-calculus (session 4): syntax, call-by-value evaluation, constants and operators
2. types, and the syntax with type annotations
3. the typing judgment and the typing rules
4. exercises

syntax of the λ-calculus (session 4, recall 1/3)

$$e ::= x \mid \lambda x \,.\, e \mid e\, e$$

- Variables: $x, y, z, f, g, \ldots$
- Abstraction (function): $\lambda x \,.\, e$
- Application: $e_1\ e_2$

- Application associates to the left: $a\ b\ c = (a\ b)\ c$
- Session 4 wrote a term t; this session writes e, for **expression**

call-by-value evaluation (session 4, recall 2/3)

a **value** is a term whose evaluation is finished

$$v ::= \lambda x \,.\, e$$

a **step** $e \to e'$ is defined by three rules (small-step operational semantics)

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2}\ \text{(E-App1)} \qquad \frac{e_2 \to e_2'}{v\, e_2 \to v\, e_2'}\ \text{(E-App2)}$$

$$\frac{}{(\lambda x \,.\, e)\, v \to e\,[x := v]}\ \text{(E-AppAbs)}$$

$$(\lambda x \,.\, x)\,(\lambda y \,.\, y) \to \lambda y \,.\, y \qquad (\lambda x \,.\, \lambda y \,.\, x)\, v_1\, v_2 \to (\lambda y \,.\, v_1)\, v_2 \to v_1$$

constants and operators (recall 3/3)

$$e ::= x \mid \lambda x\,.\,e \mid e\,e \mid n \mid b \mid e\ op\ e \qquad\qquad v ::= \lambda x\,.\,e \mid n \mid b$$

with n an integer, $b \in \{\texttt{true}, \texttt{false}\}$ and $op \in \{+, <, \dots\}$

$$\frac{e_1 \to e_1'}{e_1\ op\ e_2 \to e_1'\ op\ e_2}\ \text{(E-Op1)} \qquad \frac{e_2 \to e_2'}{v\ op\ e_2 \to v\ op\ e_2'}\ \text{(E-Op2)}$$

$$\frac{n = n_1 + n_2}{n_1 + n_2 \to n}\ \text{(E-Plus)} \qquad \frac{b = (n_1 < n_2)}{n_1 < n_2 \to b}\ \text{(E-Less)}$$

$$1 + 2 < 4 \to 3 < 4 \to \texttt{true}$$

stuck terms, neither a value nor able to step: $1 + \texttt{true}$ $\quad 1\ 2$ $\quad \texttt{true}\ (\lambda x\,.\,x)$

typing

we introduce **types**, with the following abstract syntax

τ	$::=$		**type**
	$\mid$	`int`	integer values
	$\mid$	`bool`	Boolean values
	$\mid$	$\tau \to \tau$	functions

simply-typed lambda calculus : syntax

one possibility: no annotation, the syntax as before

$$e ::= n \mid b \mid x \mid \lambda x \,.\, e \mid e\, e \mid e \;op\; e$$

the other possibility: each binder **annotated** with its type

$$e ::= n \mid b \mid x \mid \lambda x : \tau \,.\, e \mid e\, e \mid e \;op\; e$$

evaluation **ignores** the annotations: $(\lambda x : \tau \,.\, e)\, v \to e\,[x := v]$, and $\lambda x : \tau \,.\, e$ is a value

what are the implications of which version we choose?

typing judgment

the type of a variable is given by a **typing environment** Γ
(a function from variables to types)

the **typing judgment** is written

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

and reads “in typing environment Γ, expression e has type τ”

$\Gamma, x : \tau$ is Γ in which x now has the type τ (an earlier type of x is hidden)

inference rules define $\Gamma \vdash e : \tau$

typing rules: constants and variables

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \text{(T-Int)} \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}\ \text{(T-Bool)} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ \text{(T-Var)}$$

a constant **has** its type in every environment

a variable **receives** the type that the environment records for it

with $\Gamma = \{x \mapsto \texttt{int}\}$

- $\Gamma \vdash x : \texttt{int}$, by (T-Var)
- $\Gamma \vdash y : \tau$ for no type τ: no rule **applies**, since $y \notin \mathrm{dom}(\Gamma)$

typing rules: operators

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \text{ (T-Plus)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}} \text{ (T-Less)}$$

$$\frac{\dfrac{\dfrac{}{\emptyset \vdash 1 : \texttt{int}} \qquad \dfrac{}{\emptyset \vdash 2 : \texttt{int}}}{\emptyset \vdash 1 + 2 : \texttt{int}} \qquad \dfrac{}{\emptyset \vdash 4 : \texttt{int}}}{\emptyset \vdash 1 + 2 < 4 : \texttt{bool}}$$

which rule could give a type to $1 + \texttt{true}$*?*

- only (T-Plus) **concludes** on a sum
- it **requires** $\emptyset \vdash \texttt{true} : \texttt{int}$
- the only rule for `true`, (T-Bool), **gives** `bool`

$1 + \texttt{true}$ has no type

typing rules: application

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ \text{(T-App)}$$

- the function part **has** a function type $\tau_1 \to \tau_2$
- the argument **has** the type τ_1 that the function expects
- the application **receives** the type τ_2 of the result

with $\Gamma = \{f \mapsto \texttt{int} \to \texttt{bool}\}$

- $\Gamma \vdash f\ 1 : \texttt{bool}$
- f `true` has no type: f **expects** an `int`, and the only type of `true` is `bool`
- $1\ 2$ has no type: the only type of 1 is `int`, not a function type

typing rules: abstraction

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 \,.\, e) : \tau_1 \to \tau_2}\ \text{(T-Abs)}$$

in which environment is the body typed?

- in Γ **extended** with the parameter
- the annotation τ_1 **gives** the type of the parameter
- the extension **holds** for the body only: the body is the scope of x

$$\dfrac{\dfrac{}{x : \texttt{int} \vdash x : \texttt{int}} \qquad \dfrac{}{x : \texttt{int} \vdash 1 : \texttt{int}}}{\dfrac{x : \texttt{int} \vdash x + 1 : \texttt{int}}{\emptyset \vdash \lambda x : \texttt{int} \,.\, x + 1 : \texttt{int} \to \texttt{int}}}$$

exercises (1/2): derivations

1. Write typing derivation of $\emptyset \vdash (\lambda x : \texttt{bool} \,.\, x)\ \texttt{true} : \texttt{bool}$
   and $\emptyset \vdash \lambda x : \texttt{int} \,.\, \lambda y : \texttt{int} \,.\, x < y : \texttt{int} \to \texttt{int} \to \texttt{bool}$

2. Are $(\lambda x : \texttt{bool} \,.\, x)$ and $(\lambda x : \texttt{int} \,.\, x)$ the same programs?

3. Redesign typing rules for the λ-calculus without explicit type parameters. What is the type of the identity function $\lambda x \,.\, x$ now?

4. Write typing derivation for $\emptyset \vdash \lambda x \,.\, \lambda y \,.\, x\, y : \tau$, for a type τ to find

exercises (PDF)

exercises (2/2): properties of typing

5. Define type-annotation erasure function $\texttt{erase}(\cdot)$ and state its correctness theorem.

6. What does it mean for a program e to be well-typed in an empty context, i.e. $\emptyset \vdash e : \tau$ for some type τ?

7. Is there any context Γ and type τ such that $\Gamma \vdash x\, x : \tau$?

exercises (PDF)

Monomorphic typing of mini-ML

1. mini-ML: the simply typed λ-calculus with pairs, local bindings and primitives
2. its types, its typing rules, examples
3. its values and its evaluation steps
4. typing rules and type-checking algorithm; demo in OCaml

mini-ML

a function $\texttt{fun}\ x \to e$ carries **no annotation**, and a primitive stands **before its argument**: $+(x, 1)$

e	$::=$	x	identifier
	$\mid$	c	constant (1, 2, …, *true*, …)
	$\mid$	op	primitive ($+$, $\times$, *fst*, …)
	$\mid$	$\texttt{fun}\ x \to e$	function
	$\mid$	$e\ e$	application
	$\mid$	(e, e)	pair
	$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$	local binding

monomorphic typing of mini-ML

we are given **simple types**, whose abstract syntax is

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \textit{base types} \\
& \mid & \tau \to \tau & \textit{functions} \\
& \mid & \tau \times \tau & \textit{pairs}
\end{array}
$$

typing judgment

the **typing judgment** that we are going to define is written

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

and is read “in the environment Γ, the expression e has type τ ”

the environment Γ associates a type $\Gamma(x)$ with every variable x that is free in e

typing rules

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ldots \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}}\ldots$$

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

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

$\Gamma + x : \tau$ is the environment Γ' defined by $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ otherwise

example: a derivation with a local binding

$$\dfrac{\dfrac{\dfrac{\vdots}{x : \texttt{int} \vdash (x, 1) : \texttt{int} \times \texttt{int}}}{x : \texttt{int} \vdash +(x, 1) : \texttt{int}}}{\emptyset \vdash \texttt{fun}\ x \to +(x, 1) : \texttt{int} \to \texttt{int}} \qquad \dfrac{\dfrac{}{\cdots \vdash f : \texttt{int} \to \texttt{int}} \qquad \dfrac{}{\cdots \vdash 2 : \texttt{int}}}{f : \texttt{int} \to \texttt{int} \vdash f\ 2 : \texttt{int}}$$

$$\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \to +(x, 1)\ \texttt{in}\ f\ 2 : \texttt{int}$$

untypable expressions

on the other hand, we cannot type the program $1\ 2$

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

nor the program $\texttt{fun}\ x \to x\ x$

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

since $\tau_1 = \tau_1 \to \tau_2$ has no solution (types are finite)

several possible types

we can show that

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

but also

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

beware: this is not polymorphism

for a given occurrence of $\texttt{fun}\ x \to x$ we must **choose** a type

several possible types

thus, the term $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$ is not typable

because there is no type τ such that

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

on the other hand,

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

is typable (exercise: show it)

primitives

in particular, we cannot give **a single type** that is satisfactory for a primitive such as `fst`; we would have to 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.

but we can instead give **a typing rule**
for the application of `fst`:

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ e : \tau_1}$$

likewise for `snd`; $\times$ has the type of $+$

mini-ML: values and evaluation steps

$$v ::= c \mid op \mid \texttt{fun}\ x \to e \mid (v, v)$$

$$\frac{}{(\texttt{fun}\ x \to e)\ v \to e\,[x := v]} \qquad \frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \to e\,[x := v]}$$

$$\frac{n = n_1 + n_2}{+\ (n_1, n_2) \to n} \qquad \frac{}{fst\ (v_1, v_2) \to v_1} \qquad \frac{}{snd\ (v_1, v_2) \to v_2}$$

and one rule for each place where a step is taken inside a term:

- in an application, the function part; then the argument, once the function part is a value
- in a pair, the left component; then the right one, once the left one is a value
- in a local binding, the bound expression

difference between typing rules and typing algorithm

when we type `fun` $x \to e$, how do we find the type to give to x?

this is exactly the difference between the **typing rules**, which define the ternary relation

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

and the **type-checking algorithm** which checks that a given expression e is indeed typable in a given environment Γ

demo: a type checker for mini-ML

`mini_ml_typing.ml`

- the types `typ` and `expression`; the parameter of `Fun` carries its type
- the environment Γ as a persistent map (session 2)
- `type_expr`: one case for each typing rule
- programs accepted with their type, programs rejected with an error

the slides that follow show the same code

the checker: types and expressions

only the parameters of functions are annotated

```
type typ =
  | Tint
  | Tarrow    of typ * typ
  | Tproduct of typ * typ
```

```
type expression =
  | Var   of string
  | Const of int
  | Op    of string
  | Fun   of string * typ * expression
  | App   of expression * expression
  | Pair  of expression * expression
  | Let   of string * expression * expression
```

Fun **carries** the type of its parameter

the checker: the environment

the environment Γ is a persistent map from names to types (session 2)

```
module Smap = Map.Make(String)

type env = typ Smap.t
```

`Smap.add x ty env` **returns** a new map and leaves `env` unchanged: $\Gamma, x : \tau$

lookup and insertion in $O(\log n)$: the map is a balanced tree

the checker: constants, variables, primitives, pairs

```
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)
```

one case of the function for each typing rule

where is a type checked against another one? not here: these cases only **compute** a type

the checker: functions and local bindings

for a function, the type of the parameter is **given**

```
| Fun (x, ty, e) ->
    Tarrow (ty, type_expr (Smap.add x ty env) e)
```

for a local binding, it is **computed**

```
| Let (x, e1, e2) ->
    type_expr (Smap.add x (type_expr env e1) env) e2
```

the map is persistent: after the body, `env` is still the environment without `x`

the checker: applications

the only checks are in the application

```
| App (e1, e2) ->
    begin match type_expr env e1 with
    | Tarrow (ty2, ty) ->
        if type_expr env e2 = ty2 then ty
        else failwith "error: argument of wrong type"
    | _ ->
        failwith "error: function expected"
    end
```

the function part **must have** an arrow type, and the argument the type it expects

the checker: a program accepted

```
let f = fun x:int -> +(x, 1) in f 2
```

```
# typeof
    (Let ("f",
      Fun ("x", Tint, App (Op "+", Pair (Var "x", Const 1))),
      App (Var "f", Const 2)));;
```

```
- : typ = Tint
```

the checker: two programs rejected

`fun x:int -> x x`

```
# typeof (Fun ("x", Tint, App (Var "x", Var "x")));;
```

```
Exception: Failure "error: function expected".
```

`(+ 1) 2`

```
# typeof (App (App (Op "+", Const 1), Const 2));;
```

```
Exception: Failure "error: argument of wrong type".
```

Appendix

1. the typing rules of the simply typed λ-calculus, on one slide
2. static and dynamic typing in one language: Java
3. Mars Climate Orbiter: the report of the investigation board

typing rules of the simply typed λ-calculus

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \text{(T-Int)} \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}\ \text{(T-Bool)} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ \text{(T-Var)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}\ \text{(T-Plus)} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}\ \text{(T-Less)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 \,.\, e) : \tau_1 \to \tau_2}\ \text{(T-Abs)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ \text{(T-App)}$$

static and dynamic typing in one language: Java

a language may use **both** static and dynamic typing

```
Object o = "abc";
Integer n = (Integer) o;
```

accepted by the compiler: an `Object` may be an `Integer`

the cast is checked **at run time**: `ClassCastException`

```
Object[] a = new String[1];
a[0] = 42;
```

accepted by the compiler: an array of `String` is an array of `Object`

the store is checked **at run time**: `ArrayStoreException`

each object carries its class at run time, so the check is possible

Mars Climate Orbiter: the report of the investigation board

Mars Climate Orbiter Mishap Investigation Board, Phase I Report, NASA, 10 November 1999

- page 6: the signal was “last seen at approximately 09:04:52 UTC on Thursday, September 23, 1999”
- page 13: the thruster data were “delivered in lb-sec instead of the specified and expected units of Newton-sec”
- page 13: the computed effect on the trajectory was “low by a factor of 4.45 (1 pound force=4.45 Newtons)”; estimated closest approach 226 km, reconstructed 57 km, “judged too low for spacecraft survival”
- page 16: “The Angular Momentum Desaturation (AMD) file contained the output data from the SM_FORCES software”

the report prints no source code

`llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf`