# Types and safety

C. Paulin (courtesy of T. Balabonski)

M1 MPRI 2025–26

# Where are we ?

Abstract syntax trees for the FUN language

```
type bop = Add | Sub | Mul | Lt | Eq
type expr =
  | Int of int
  | Bool of bool
  | Bop of bop * expr * expr
  | Var of string
  | Let of string * expr * expr
  | If  of expr * expr * expr
  | App of expr * expr
  | Fun of string * expr
  | Fix of string * expr
```

# Where are we ? Big step semantics

Specifies the interpreter
Two possible choices for values :

1. explicit environments : integers, booleans and closures
2. substitution : integer, booleans and abstractions (functions)

Rules

$$\frac{}{n \Longrightarrow n}$$

$$\frac{e_1 \Longrightarrow n_1 \qquad e_2 \Longrightarrow n_2}{e_1 \oplus e_2 \Longrightarrow n_1 + n_2}$$

$$\frac{e_1 \Longrightarrow v_1 \qquad e_2[x := v_1] \Longrightarrow v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Longrightarrow v}$$

$$\frac{}{\texttt{fun } x \texttt{ -> } e \Longrightarrow \texttt{fun } x \texttt{ -> } e}$$

$$\frac{e_1 \Longrightarrow \texttt{fun } x \texttt{ -> } e \qquad e_2 \Longrightarrow v_2 \qquad e[x := v_2] \Longrightarrow v}{e_1\ e_2 \Longrightarrow v}$$

# Where are we ? Small step semantics

Describes how the computation is done

$$\frac{e_1 \to e'_1}{e_1 \oplus e_2 \to e'_1 \oplus e_2} \qquad \frac{e_2 \to e'_2}{v_1 \oplus e_2 \to v_1 \oplus e'_2} \qquad \frac{n_1 + n_2 = n}{n_1 \oplus n_2 \to n}$$

$$\frac{e_1 \to e'_1}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to \texttt{let}\ x\ \texttt{=}\ e'_1\ \texttt{in}\ e_2}$$

$$\frac{}{\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e \to e[x := v]}$$

$$\frac{e_1 \to e'_1}{e_1\ e_2 \to e'_1\ e_2} \qquad \frac{e_2 \to e'_2}{v_1\ e_2 \to v_1\ e'_2} \qquad \frac{}{(\texttt{fun}\ x\ \texttt{->}\ e)\ v \to e[x := v]}$$

# Our goal : MiniML

Design an interpreter (and then a compiler) for a more realistic functional language (MiniML)

- More operators on basic types
- Pairs
- Algebraic types and pattern-matching

Today :

- type-checking (theory and practice)
- implementation of type-checking and interpreter for MiniML

# Objectives

- Another semantics of the FUN language
- Classification of the various kinds of values a program may deal with
- Make sure the program handles the data in a consistent way
- Reject inconsistent programs (before execution if possible)

# Representation of data

- Data stored as a sequence of bits
- Example of a 32-bits memory word

  `1110 0000 0110 1100 0110 0111 0100 1000`

  Hexadecimal representation
  $(0-9\ a = 10 = (1010)_2\ b = 11 = (1011)_2$
  $c = 12 = (1100)_2, \ldots f = 15 = (1111)_2$

  `0x e0 6c 67 48`

- The meaning of such a word depends on the context
  - memory address : 3 765 200 712,
  - a 32-bit signed integer in 2's complement : $-529\,766\,584$,
  - a simple precision floating point number (IEEE754 standard) : $15\,492\,936 \times 2^{42}$,
  - a character string in Latin-1 encoding : "Holà".

# Inconsistent operations

All operations in a programming language are constrained.
In caml for instance,

- the addition 5 + 37 of two integers is possible,
- but the operations "5" + 37, 5 + (**fun** x -> 37) or 5(37) are not.

In our interpreter, we classified the possible values

```
type value =
  | VInt  of int
  | VBool of bool
  | VClos of string * expr * env
```

The interpreter checked consistency between values and operators

```
let rec eval e env = match e with
  | Bop(op, e1, e2) ->
    begin match op, eval e1 env, eval e2 env with
      | Add, VInt n1, VInt n2 -> VInt (n1 + n2)
      ...
      | _ -> failwith "unauthorized operation"
    end
```

# Types : a classification of values

Programming languages usually distinguish numerous kinds of values, called types.

Basic types :

- numbers : `int` , `double`,
- booleans : `bool`,
- characters : `char`,
- character strings : `string` .

Richer types built over these base types.

- arrays : `int []` ,
- functions : `int -> bool`,
- data structures : `struct point { int x; int y; };`,
- objects : `class Point { public final int x, y; ... }`.

Once this classification is set, each operation is defined to apply to elements of some given type.

# Overloading

overloading : one operator may be applied to various types of elements, with different meanings depending on the type.
For instance, in python or java the operator + may apply :

- to two integers, in which case it denotes an addition : 5 + 37 = 42,
- to two strings, in which case it denotes a concatenation :
  "5" + "37" = "537".

# Casting

casting : converting (implicitely or not) a value of some type to another type.
The operation "5" + 37 mixing a string and an integer may evaluate to :

- 42 in php, where the string "5" is converted into the number 5,
- "537" in java, where the integer 37 is converted into the string "37".

Note that such a conversion requires an actual modification of the data !

- the number 37 is represented by 0x 00 00 00 25
- the string "37" by 0x 00 00 37 33.

# Summary on types

- The type of a value gives the key for interpreting the associated data.
- It may be required for selecting the appropriate operations.
- Inconsistent types are likely to reveal programming errors (and programs that should not be executed).

# Static versus dynamic type analysis

Handling types at runtime is costly in several ways :

- some memory has to be used to pair each data with an identification of its type,
- runtime tests are necessary to select the operations to apply to the data,
- execution may be interrupted when a type error appears, ...

In dynamically typed languages such as python, theses costs are paid in full. Conversely, statically typed languages such as C, java or caml save us at least a part of this runtime cost, since types are handled at compile time.

# Static type analysis

- Static type analysis : type analysis is performed at compilation time
- It consists in associating with each expression in a program a type, which predicts the type of the value that will be obtained when evaluating the expression.
- This prediction is based on constraints given by each constructor of the abstract syntax.

# Static type analysis : examples

$e1 \oplus e2$

- the expression will produce an integer,
- for the operation to be consistent, both subexpressions e1 and e2 must also produce integers.

**let** x = e **in** x + 1

- We associate to each variable the type of the value the variable refers to.
- The type of x is the type of the value produced by the expression e, we expect it to be the type of integers.

**fun** x –> x + 1

- the type of a function makes the expected types of all parameters explicit as well as the type of the result

# Type safety

This verification of type consistency before the execution of a program is associated to the idea, formulated by Robin Milner, that

Well-typed programs do not go wrong.

- static type analysis : reject absurd programs before they are ever executed (or released to clients...)
- limits :
  - type checking performed at compile time should be decidable (and reasonably fast)
  - identifying exactly buggy programs (like non-terminating ones) is usually undecidable (or costly)

Compromise

- give some safety, by rejecting many absurd programs,
- and let to programmers enough expressiveness, by not rejecting too many non-absurd programs.

# Type annotation

Type analysis may require some amount of annotations from the programmer.

1. Annotate each subexpression, the compiler just checks consistency.

```
fun (x : int) ->
  let (y : int) = ((x : int) + (1 : int) : int)
  in (y : int)
```

2. Annotate only variables, and formal parameters of functions (C or Java). The compiler deduces the type of each expression.

```
fun (x : int) -> let (y : int) = x+1 in y
```

3. Annotate only function parameters (sufficent for typechecking MiniML).

```
fun (x : int) -> let y = x+1 in y
```

4. No annotation (Caml).

```
fun x -> let y = x+1 in y
```

In this last case, the compiler must infer the type of each variable and expression, with no help from the programmer.

# Advantage of static type-checking

- selecting the appropriate operation for overloaded operators is done at compilation time, and costs nothing at execution time.
- checking the consistency of types at compilation time allow early detection of many program inconsistencies, and consequently early correction of bugs.

Next

- formalize the notion of type and the associated constraints for FUN
- implement type checking and type inference
- formally state and prove that well-typed programs does not go wrong

# Types

1. Types and safety
   - Types values and operations
   - Typing judgment and inference rules
   - Type safety
   - Type verification for FUN
   - Polymorphism
   - Type inference

# Typing judgement

Well-typed programs are characterized by a set of rules that allow justififying that “in some context $\Gamma$, an expression $e$ is consistent and has type $\tau$”.
This sentence is called a typing judgment, and is written

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

The context $\Gamma$ in a typing judgment maps a type to each variable of the expression $e$.
The typing judgment is not function but a relation between three elements : context, expression, type.

- some expressions $e$ have no type (because they are inconsistent),
- in some situations, several types might be possible for a given expression and context.

# Typing rules : consistency and types

Type of an expression for a fragment of the FUN language :

$$
\begin{array}{rcl}
e & ::= & n \\
 & | & e \texttt{ + } e \\
 & | & x \\
 & | & \texttt{let } x \texttt{ = } e \texttt{ in } e \\
 & | & \texttt{fun } x \texttt{ -> } e \\
 & | & e\, e
\end{array}
$$

We need a base type for numbers, as well as function types.

$$
\begin{array}{rcl}
\tau & ::= & \texttt{int} \\
 & | & \tau \to \tau
\end{array}
$$

A type of the form $\tau_1 \to \tau_2$ is the type of a function that expects a parameter of type $\tau_1$ and returns a result of type $\tau_2$.

# Typing rules

We associate to each construction of the language a rule giving :

- the type such expression may have, and
- the constraints that have to be satisfied for the expression to be consistent.

# Inference rules

- An integer constant $n$ has the type int.

$$\frac{}{\Gamma \vdash n : \texttt{int}}$$

- If both expressions $e_1$ and $e_2$ are consistent and of type int, then the expression $e_1 \texttt{ + } e_2$ is consistent, also with type int.

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

- A variable has the type given by the environment.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} x \in \mathrm{dom}(\Gamma)$$

- A local variable is associated to the type of the expression that defines it.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2}$$

# Inference rules : functions

- In the body of a function, the formal parameter is seen as an ordinary variable, whose type corresponds to the expected type of the parameter.

$$\frac{\Gamma, x : \tau_1 \;\vdash\; e \;:\; \tau_2}{\Gamma \;\vdash\; \texttt{fun}\;\; x \;\; \texttt{->} \;\; e \;:\; \tau_1 \to \tau_2}$$

- A function has to be applied to an argument of the expected type.

$$\frac{\Gamma \;\vdash\; e_1 \;:\; \tau_2 \to \tau_1 \qquad \Gamma \;\vdash\; e_2 \;:\; \tau_2}{\Gamma \;\vdash\; e_1\; e_2 \;:\; \tau_1}$$

# Inference rules for simple types : summary

The simple types for our fragment of the FUN language are fully defined by the six following inference rules.

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

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

# Typable expressions

A typing judgment is justified by a series of deductions obtained by applying the inference rules.
For instance, given the context $\Gamma = \{\, x : \texttt{int},\ f : \texttt{int} \to \texttt{int} \,\}$ we may reason as follow.

1. $\Gamma \vdash x : \texttt{int}$ is valid, by the rule on variables.
2. $\Gamma \vdash f : \texttt{int} \to \texttt{int}$ is valid, by the rule on variables.
3. $\Gamma \vdash \texttt{1} : \texttt{int}$ is valid, by the rule on constants.
4. $\Gamma \vdash f\ \texttt{1} : \texttt{int}$ is valid, by the rule on application, using the already justified points 2. and 3.
5. $\Gamma \vdash x + f\ \texttt{1} : \texttt{int}$ is valid, by the rule on addition, using 1. et 4.

This reasoning is called a derivation, it can be represented as a derivation tree whose root is the conclusion we want to justify.

$$\dfrac{\dfrac{}{\Gamma \vdash x : \texttt{int}} \qquad \dfrac{\dfrac{}{\Gamma \vdash f : \texttt{int} \to \texttt{int}} \qquad \dfrac{}{\Gamma \vdash \texttt{1} : \texttt{int}}}{\Gamma \vdash f\ \texttt{1} : \texttt{int}}}{\Gamma \vdash x + f\ \texttt{1} : \texttt{int}}$$

# Multiple derivations

$$\vdash \texttt{fun x -> x} : \texttt{int} \rightarrow \texttt{int}$$

$$\vdash \texttt{fun x -> x} : (\texttt{int} \rightarrow \texttt{int}) \rightarrow (\texttt{int} \rightarrow \texttt{int})$$

are both valid. Here, the absence of the context $\Gamma$ means that we consider the empty context.

# Untypable expressions

- We may need to prove that an expression has no type :
  no judgment $\Gamma \vdash e : \tau$ can be justified using the typing rules
- Show that any attempt at building a typing tree for the expression necessarily fails
- Inversion properties for the typing judgement
  - Given the form of the judgement, the only rule(s) which applies is ... and the the premisses hould also be derivable

# Untypable expressions : examples

(5 37),

- only the application rule can lead to a judgment $\Gamma \vdash \texttt{5 37} : \tau$,
- the two premises $\Gamma \vdash \texttt{5} : \tau' \to \tau$ and $\Gamma \vdash \texttt{37} : \tau'$ should be derivable
- no rule allows giving a functional type to an integer constant (the only rule would give the typing judgment $\Gamma \vdash \texttt{5} : \texttt{int}$).

**fun** x –> x x.

- a derivation tree for a judgment $\Gamma \vdash \texttt{fun x -> x x} : \tau$ would necessarily have the shape

$$\dfrac{\dfrac{\Gamma, \texttt{x} : \tau_1 \vdash \texttt{x} : \tau_1 \to \tau_2 \qquad \Gamma, \texttt{x} : \tau_1 \vdash \texttt{x} : \tau_1}{\Gamma, \texttt{x} : \tau_1 \vdash \texttt{x x} : \tau_2}}{\Gamma \vdash \texttt{fun x -> x x} : \tau_2}$$

- the premise $\Gamma, \texttt{x} : \tau_1 \vdash \texttt{x} : \tau_1 \to \tau_2$ cannot be justified : the inference rules allow only the type $\tau_1$ for $\texttt{x}$, and there are no (finite) types $\tau_1$ and $\tau_2$ such that $\tau_1 = \tau_1 \to \tau_2$.

# Reasoning on well-typed expressions

Prove that for any context $\Gamma$, any expression $e$ and any type $\tau$ :
if $\Gamma \vdash e : \tau$ is valid then all the free variables of $e$ are in the domain of $\Gamma$.

- By induction on the structure of the typing derivation tree.
- We have one proof case for each inference rule,
- each premise of the rule yields an induction hypothesis.

# Proof (1/2)

Property which depends on $\Gamma$, $e$ and $\tau$ : $\P(\Gamma, e, \tau) \stackrel{\text{def}}{=} \mathrm{fv}(e) \subseteq \mathrm{dom}(\Gamma)$
Proof by induction on $\Gamma \vdash e : \tau$.

- Case $\Gamma \vdash n : \texttt{int}$. We have $\mathrm{fv}(n) = \emptyset$, and of course $\emptyset \subseteq \mathrm{dom}(\Gamma)$.
- Case $\Gamma \vdash x : \Gamma(x)$. We have $\mathrm{fv}(x) = \{ x \}$, and the application of the rule indeed assumes $x \in \mathrm{dom}(\Gamma)$.
- Case $\Gamma \vdash e_1 + e_2 : \texttt{int}$, with premises $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$. The premises give two induction hypotheses $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma)$. By definition of free variables we have $\mathrm{fv}(e_1 + e_2) = \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$. With the induction hypotheses we deduce that $\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma)$, and therefore $\mathrm{fv}(e_1 + e_2) \subseteq \mathrm{dom}(\Gamma)$.

# Proof (2/2)

- Case $\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ :\ \tau_2$, with premises $\Gamma \vdash e_1 : \tau_1$ and $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$. The premises give two induction hypotheses $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma) \cup \{ x \}$ (note that the premise related to the judgment $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$ mentions an environment extended with the variable $x$). By definition we have $\mathrm{fv}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) = \mathrm{fv}(e_1) \cup (\mathrm{fv}(e_2) \setminus \{ x \})$. The first induction hypothesis ensures that $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$. The second induction hypothesis ensures that $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma) \cup \{ x \}$, from which we deduce $\mathrm{fv}(e_2) \setminus \{ x \} \subseteq \mathrm{dom}(\Gamma)$. Therefore we have $\mathrm{fv}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) \subseteq \mathrm{dom}(\Gamma)$.
- Both cases related to functions are similar to the cases above.

# Full rules for FUN

New base type `bool` for boolean values : $\tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau \rightarrow \tau$
Three additional typing rules for the missing constructs.

- An expression $e_1 < e_2$ has type `bool` whenever $e_1$ and $e_2$ are numbers.

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

- A conditional requires the condition to be a boolean and the two branches to have the same type

$$\frac{\Gamma \vdash c : \mathtt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}\ c\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

- A recursive expression must have the same type $\tau$ as the recursive references it contains.
  If the expression $e$ has type $\tau$, in an environment where the identifier $x$ also has this type $\tau$, then the expression $\mathtt{fix}\ x = e$ is consistent of type $\tau$.

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{fix}\ x = e : \tau}$$

# Typing and big step semantics

Using the notion of natural semantics, we can prove the following statement relating the typing and the evaluation of an expression.

$$\text{If} \quad \Gamma \vdash e : \tau \quad \text{and} \quad e \Longrightarrow v \quad \text{then} \quad \Gamma \vdash v : \tau.$$

This means that the evaluation relation preserves the consistency and the types of expressions.

- However, we assume that the evaluation is indeed possible and reaches a value.
- It does not prove that the evaluation of well-typed programs indeed produce a value
- It says nothing about programs that break or loop.
- We need the small step semantics to get an actual safety property.

# Type safety, small step version

With a reduction semantics, the safety property may be state as : the evaluation of a well-typed program never blocks on an inconsistent operation. We formalize the property through two lemmas.

- Progress lemma : a well-typed expression is never blocked.

  If $\Gamma \vdash e : \tau$ then $e$ is a value or there is $e'$ such that $e \to e'$.

- Type preservation lemma : reduction preserves types.

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

  Historically, the type preservation lemma was called subject reduction

# Consequence of progress and type preservation

Starting with a well-typed expression $e_1$ with type $\tau$ :

- if $e_1$ is not already a value, then it reduces to $e_2$, which is still well-typed with type $\tau$ and thus, in case it is not a value, reduces in turn to $e_3$ well-typed of type $\tau$, on so on.

$$(e_1 : \tau) \;\rightarrow\; (e_2 : \tau) \;\rightarrow\; (e_3 : \tau) \;\rightarrow\; \ldots$$

- At the far right side of this sequence, there are two possible scenarios :
  - either we reach a value $v$ (still well-typed with type $\tau$),
  - or the reduction go on infinitely.
  - The reduction cannot end with a blocked expression.

# Progress

If $\Gamma \vdash e : \tau$, then $e$ is a value, or there is $e'$ such that $e \to e'$.
We consider the simple types for the fragment of the FUN language defined by the following rules.

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

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

# Proof of progress

We will prove the lemma by induction on the derivation of $\Gamma \vdash e : \tau$.

- Case $\Gamma \vdash n : \mathtt{int}$. Then $n$ is a value.
- Case $\Gamma \vdash \mathtt{fun}\ x\ \mathtt{->}\ e : \tau_1 \to \tau_2$. Then $\mathtt{fun}\ x\ \mathtt{->}\ e$ is a value.
- Case $\Gamma \vdash e_1\ e_2 : \tau_1$, with $\Gamma \vdash e_1 : \tau_2 \to \tau_1$ and $\Gamma \vdash e_2 : \tau_2$. Induction hypotheses give us the two following disjunctions.
  1. $e_1$ is a value or $e_1 \to e_1'$,
  2. $e_2$ is a value or $e_2 \to e_2'$.

  We reason by case on these disjunctions.
  - If $e_1 \to e_1'$, then $e_1\ e_2 \to e_1'\ e_2$ : goal completed.
  - Otherwise, $e_1$ is a value $v_1$.
    - If $e_2 \to e_2'$, then $v_1\ e_2 \to v_1\ e_2'$ : goal completed.
    - Otherwise, $e_2$ is a value $v_2$. Since we have as hypothesis the typing judgment $\Gamma \vdash v_1 : \tau_2 \to \tau_1$, we know that $v_1$ necessarily has the shape $\mathtt{fun}\ x\ \mathtt{->}\ e$ (classification lemma detailed below). Then we have

      $$e_1\ e_2 = (\mathtt{fun}\ x\ \mathtt{->}\ e)\ v_2 \to e[x := v_2]$$

      which completes our case.
- Other cases are similar.

# Classification lemma for typed values

Let $v$ be a value such that $\Gamma \vdash v : \tau$. Then :

- if $\tau = \mathtt{int}$, then $v$ has the shape $n$,
- if $\tau = \tau_1 \rightarrow \tau_2$, then $v$ has the shape $\mathtt{fun}\ x\ \mathtt{->}\ e$.

Proof by case on the last rule applied in the derivation of $\Gamma \vdash v : \tau$, knowing that the only two possible shapes for a value are : $n$ or $\mathtt{fun}\ x\ \mathtt{->}\ e$.

# Proof of type Preservation

If $e \to e'$ then for all $\tau$, if $\Gamma \vdash e : \tau$ then $\Gamma \vdash e' : \tau$.
Proof by induction on the derivation of $e \to e'$.

- Case $n_1 + n_2 \to n$ with $n = n_1 + n_2$. The hypothesis $\Gamma \vdash n_1 + n_2 : \tau$ implies $\tau = \texttt{int}$ (inversion).
  Moreover $\Gamma \vdash n : \texttt{int}$
- Case $e_1 + e_2 \to e_1' + e_2$ with $e_1 \to e_1'$.
  Induction hypothesis "if $\Gamma \vdash e_1 : \tau'$, then $\Gamma \vdash e_1' : \tau'$".
  The hypothesis $\Gamma \vdash e_1 + e_2 : \tau$ implies $\tau = \texttt{int}$, $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$ (inversion lemma).
  Thus by induction hypothesis $\Gamma \vdash e_1' : \texttt{int}$ and

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

# Proof of type Preservation (continued)

- Case $(\texttt{fun}\ x\ \texttt{in}\ e)\ v \rightarrow e[x := n]$.
  From $\Gamma \vdash (\texttt{fun}\ x\ \texttt{->}\ e)\ v : \tau$ we know there is $\tau'$ such that $\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau' \rightarrow \tau$ and $\Gamma \vdash v : \tau'$ and from $\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau' \rightarrow \tau$ we further deduce $\Gamma, x : \tau' \vdash e : \tau$ (inversion lemma).
  We have $\Gamma, x : \tau' \vdash e : \tau$ and $\Gamma \vdash v : \tau'$, from which we deduce $\Gamma \vdash e[x := v] : \tau$ using a substitution lemma.
- The other cases are similar.

# Inversion lemma

- If $\Gamma \vdash e_1 \texttt{ + } e_2 : \tau$ then $\tau = \texttt{int}$, $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$.
- If $\Gamma \vdash e_1\ e_2 : \tau$ then there is $\tau'$ such that $\Gamma \vdash e_1 : \tau' \to \tau$ and $\Gamma \vdash e_2 : \tau'$.
- If $\Gamma \vdash \texttt{fun}\ x \texttt{ -> } e : \tau$ then there are $\tau_1$ and $\tau_2$ such that $\tau = \tau_1 \to \tau_2$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$.

<u>Proof by case on the last rule of the typing derivation.</u>

# Substitution lemma

Replacing a typed variable by an identically typed expression preserves typing.

$$\text{If} \quad \Gamma, x : \tau' \vdash e : \tau \quad \text{and} \quad \Gamma \vdash e' : \tau' \quad \text{then} \quad \Gamma \vdash e[x := e'] : \tau.$$

Proof by induction on the derivation of $\Gamma, x : \tau' \vdash e : \tau$.

# Type safety theorem

The following theorem combines the progress lemma and the type preservation lemme.

If $\Gamma \vdash e : \tau$ and $e \rightarrow^* e'$ with $e'$ not reducible, then $e'$ is a value.

The proof is by recurrence on the length of the reduction sequence $e \rightarrow^* e'$.

# Summary

- The safety property of typed expressions establishes a link between a static property (type consistency) and a dynamic property (evaluation without errors) of programs.
- It is still possible that a well-typed program fails to reach a value, in case the evaluation never ends.
- More generally, programming languages with a strict typing discipline are able to detect many errors early (at compilation time), which results in less errors at execution time.

# Types

# Writing a type-checker

- We use caml to write a type checker from FUN programs with enough annotations
- We follow the typing rules given in the previous sections
- This program consists in a function type_expr, which takes as parameters an expression $e$ and an environment $\Gamma$ and which :
  - returns the unique type that can be associated to $e$ in the environment $\Gamma$ if $e$ is indeed consistent in this environment,
  - fails otherwise.

# Data types for type-checking

(caml) datatype to represent the types of the FUN language :

```
type typ = | TInt | TBool | TFun of typ * typ
```

Abstract syntax trees of the FUN language now include some type annotations in functions and fixpoints.

```
type bop = Add | Sub | Mul | Lt | Eq
type expr =
  ...
  | Fun of string * typ * expr
  | Fix of string * typ * expr
```

The environment is an association table between variable identifiers (string) and FUN types.

```
module Env = Map.Make(String)
type type_env = typ Env.t
```

# The type-checking function

```
let rec type_expr (e:expr) (env : type_env) : typ =
  match e with
    | Int _ -> TInt
    | Var(x) -> Env.find x env
    | Bop(Add, e1, e2) ->
       let t1 = type_expr e1 env in
       let t2 = type_expr e2 env in
       if t1 = TypInt && t2 = TypInt then TypInt
       else failwith "type␣error"
    | If(c, e1, e2) ->
       let tc = type_expr c env in
       let t1 = type_expr e1 env in
       let t2 = type_expr e2 env in
       if tc = TBool && t1 = t2 then t1
       else failwith "type␣error"
```

# The type-checking function

```
| Let(x, e1, e2) ->
  let t1 = type_expr e1 env in
  type_expr e2 (Env.add x t1 env)
| Fun(x, tx, e) ->
  let te = type_expr e (Env.add x tx env) in
  TFun(tx, te)
| App(f, a) ->
  let tf = type_expr f env in
  let ta = type_expr a env in
  begin match tf with
    | TFun(tx, te) ->
      if tx = ta then te
      else failwith "type␣error"
    | _ -> failwith "type␣error"
    end
```

# The type-checking function

```
| Fix(f, t, e) ->
   let env' = Env.add f t env in
   let te = type_expr e env' in
   if te = t then t
   else failwith "type␣error"
```

# Limit of simple types

With the simple types, an expression such as

```
fun x -> x
```

may have several distinct types (same code works for different sort of datas).
However, it can have only one type at a time.
In particular, in an expression such as

```
let f = fun x -> x in f f
```

we have to choose only one type for the f, and the expression cannot be typed.
This is called a monomorphic type (literally : one shape).

# Parametric polymorphism

- The possibility of using parametrized types, which cover many variants of a given shape of type.
- We extend the grammar of the types $\tau$ with two new elements :
  - type variables, or type parameters, written $\alpha$, $\beta$, ... denoting indeterminate types,
  - a universal quantification $\forall\alpha.\tau$ denoting a polymorphic type, where the type variable $\alpha$ may, in $\tau$, denote any type.

# Polymorphic types in FUN

The set of types is then defined by the extended grammar

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \\ & | & \tau \rightarrow \tau \\ & | & \alpha \\ & | & \forall \alpha . \tau \end{array}$$

# Instantiation

If an expression *e* has a polymorphic type $\forall\alpha.\tau$, then for any type $\tau'$ we can consider *e* to also be of type $\tau[\alpha := \tau']$

$$\frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e : \tau[\alpha := \tau']}$$

The notion of type substitution $\tau[\alpha := \tau']$ is defined by a set of equations.

$$\begin{array}{rcl}
\mathtt{int}[\alpha := \tau'] & = & \mathtt{int} \\
\beta[\alpha := \tau'] & = & \left\{ \begin{array}{ll} \tau' & \text{if } \alpha = \beta \\ \beta & \text{if } \alpha \neq \beta \end{array} \right. \\
(\tau_1 \rightarrow \tau_2)[\alpha := \tau'] & = & \tau_1[\alpha := \tau'] \rightarrow \tau_2[\alpha := \tau'] \\
(\forall\beta.\tau)[\alpha := \tau'] & = & \left\{ \begin{array}{ll} \forall\beta.\tau & \text{if } \alpha = \beta \\ \forall\beta.\tau[\alpha := \tau'] & \text{if } \alpha \neq \beta \text{ and } \beta \notin \mathrm{fv}(\tau') \end{array} \right.
\end{array}$$

The notion of free type variable is also defined similarly.

$$\begin{array}{rcl}
\mathrm{fv}(\mathtt{int}) & = & \emptyset \\
\mathrm{fv}(\alpha) & = & \{\,\alpha\,\} \\
\mathrm{fv}(\tau_1 \rightarrow \tau_2) & = & \mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2) \\
\mathrm{fv}(\forall\alpha.\tau) & = & \mathrm{fv}(\tau) \setminus \{\,\alpha\,\}
\end{array}$$

# Generalisation

When an expression has a type $\tau$ containing a parameter $\alpha$, and this parameter is not constrained in any way by the context $\Gamma$ (does not appear in $\Gamma$), then we can consider $e$ as a polymorphic expression, with type $\forall\alpha.\tau$.

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\tau}$$

Formally, the set of free type variables of an environment $\Gamma = \{\, x_1 : \tau_1, \ldots, x_n : \tau_n \,\}$ is defined by :

$$\mathrm{fv}(\{\, x_1 : \tau_1, \ldots, x_n : \tau_n \,\}) \quad = \quad \bigcup_{1 \leq i \leq n} \mathrm{fv}(\tau_i)$$

# Examples and counter-examples

We can now give to the identity function **fun** x –> x the polymorphic type $\forall\alpha.\alpha \to \alpha$, stating that this function takes an argument of any type and returns a result of the same type.

$$\dfrac{\dfrac{\dfrac{}{\texttt{x} : \alpha \vdash \texttt{x : } \alpha}}{\vdash \texttt{fun x -> x : } \alpha \to \alpha} \qquad \alpha \notin \mathsf{fv}(\emptyset)}{\vdash \texttt{fun x -> x : } \forall\alpha.\alpha \to \alpha}$$

The key here is that **fun** x –> x can have the type $\alpha \to \alpha$ in the empty context, and that the empty context, in particular, puts no constraint on $\alpha$.

# Example

It is then possible to type the expression **let** f = **fun** x –> x **in** f f.
We type the expression f f, in an environment $\Gamma = \{ f : \forall\alpha.\alpha \to \alpha \}$ with which we can complete the derivation as follows.

$$\dfrac{\dfrac{\overline{\Gamma \vdash \texttt{f} : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})} \qquad \dfrac{\overline{\Gamma \vdash \texttt{f} : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : \texttt{int} \to \texttt{int}}}{\Gamma \vdash \texttt{f}\ \texttt{f} : \texttt{int} \to \texttt{int}}$$

# Other solution

Note that this is not the only solution :

- we can replace the concrete type int by a type variable $\beta$,
- the resulting type could be generalized

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathtt{f} : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \mathtt{f} : (\beta \to \beta) \to (\beta \to \beta)} \qquad \dfrac{\dfrac{}{\Gamma \vdash \mathtt{f} : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \mathtt{f} : \beta \to \beta}}{\Gamma \vdash \mathtt{f}\ \mathtt{f} : \beta \to \beta} \quad \beta \notin \mathrm{fv}(\Gamma)$$

$$\dfrac{\Gamma \vdash \mathtt{f}\ \mathtt{f} : \beta \to \beta \qquad \beta \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash \mathtt{f}\ \mathtt{f} : \forall\beta.\beta \to \beta}$$

# Counter-example

- This system however <u>does not</u> allow the type $\alpha \to \forall\alpha.\alpha$ for the identity function **fun** x –> x.
- Indeed, this would require giving to *x* the type $\forall\alpha.\alpha$ in a context $\Gamma = \{\,\mathtt{x} : \alpha\,\}$.
- Our axiom rule only allows the derivation of $\Gamma \vdash \mathtt{x} : \alpha$, $\alpha$ cannot be generalized, since it appears in $\Gamma$.
- Thus we (fortunately) cannot use polymorphic types to allow the ill-formed expression (**fun** x –> x) 5 37.

# Example : composition function

Let us show that composition function **fun** f –> **fun** g –> **fun** x –> g (f x) has the polymorphic type $\forall\alpha\beta\gamma.(\alpha\to\beta)\to(\beta\to\gamma)\to(\alpha\to\gamma)$.
Write Γ the environment $\{\,\texttt{f} : \alpha\to\beta, \texttt{g} : \beta\to\gamma, \texttt{x} : \alpha\,\}$.
We can build the following derivation :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{g} : \beta\to\gamma} \qquad \dfrac{\dfrac{}{\Gamma\vdash \texttt{f} : \alpha\to\beta}\qquad \dfrac{}{\Gamma\vdash \texttt{x} : \alpha}}{\Gamma\vdash \texttt{f x} : \beta}}{\Gamma \vdash \texttt{g (f x)} : \gamma}}{\vdash \texttt{fun f -> fun g -> fun x -> g (f x)} : (\alpha\to\beta)\to(\beta\to\gamma)\to(\alpha\to\gamma)} \qquad \alpha,\beta,\gamma\notin \mathsf{fv}(\emptyset)}{\vdash \texttt{fun f -> fun g -> fun x -> g (f x)} : \forall\alpha\beta\gamma.(\alpha\to\beta)\to(\beta\to\gamma)\to(\alpha\to\gamma)}$$

Exercise : show that this composition function can also have the type $\forall\alpha\beta.(\alpha\to\beta)\to\forall\gamma.(\beta\to\gamma)\to(\alpha\to\gamma)$.

# Undecidability of type-checking

Without annotations from the programmer, the two following questions about polymorphic types in FUN are undecidable :

- type inference : an expression $e$ being given, determine whether there is a type $\tau$ such that $\Gamma \vdash e : \tau$ (and provide the type),
- type verification : an expression $e$ and a type $\tau$ being given, determine whether $\Gamma \vdash e : \tau$.

These undecidability results still hold for any language extending the FUN kernel.

# Restricting polymorphism

- To check the type consistency of a program, or infer the type of a program, we have to either require some amount of annotations or restrict the use of polymorphism.
- Each language sets is own balance between the amount of annotation and the expressiveness of the type system.
- In caml, polymorphism is restricted by a simple fact :
  - we cannot write any explicit quantifier in a type.
  - every type variable that is globally free is implicitly considered to be universally quantified.

# Example

The caml type for the first projection of a pair

```
fst: 'a * 'b -> 'a
```

is actually the generalized type $\forall \alpha\beta.\alpha \times \beta \to \alpha$.
Similarly, the caml type for the left iterator of a list

```
List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

has to be understood as $\forall \alpha\beta.(\alpha \to \beta \to \alpha) \to \alpha \to \beta\,\texttt{list} \to \alpha$.

# Hindley-Milner system

- This restricted polymorphism is common to all languages of the ML family, and called the Hindley-Milner system.
- It only allows “prenex” quantification
- It distinguishes the notion of type $\tau$ without quantification, and the notion type scheme $\sigma$ which is a type extended with global quantifiers.

# Hindley-Milner system for FUN

For our fragment of FUN, this can be described by the following grammar.

$$
\begin{array}{rcl}
\tau & ::= & \mathtt{int} \\
& \mid & \tau \rightarrow \tau \\
& \mid & \alpha \\
\\
\sigma & ::= & \forall\alpha_1 \ldots \forall\alpha_n.\tau
\end{array}
$$

In this system, we can work with type schemes such as $\forall\alpha.\alpha \rightarrow \alpha$ and $\forall\alpha\beta\gamma.(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \gamma)$, but we cannot express a type with the shape $(\forall\alpha.\alpha \rightarrow \alpha) \rightarrow (\forall\alpha.\alpha \rightarrow \alpha)$.
In the Hindley-Milner system, we adapt contexts and typing judgments to allow the association of a type scheme to a variable or an expression :

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e : \sigma$$

Note that a type scheme with zero quantifier is just a type.

# Typing rules

Typing rules are also adapted in a way such that type schemes are authorized only at some specific places.

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

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \sigma_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

$$\frac{\Gamma \vdash e : \forall \alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha := \tau]} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha.\sigma}\alpha \notin \mathsf{fv}(\Gamma)$$

# Typing rules

The generalization of the type of an expression is only allowed at two places :

- at the root of the program,
- for the argument of a `let` definition.
- the typing rule for `let` contains type schemes
- the type of an application requires both the type of the function and the type of its arguments to be simple types.

The Hindley-Milner type system has two notable properties :

- type checking <u>and</u> type inference are decidable (see next section),
- the system ensures type safety : the evaluation of the well-typed program cannot be stopped by an inconsistent operation (the proof extends the one already given for simple types).

# Types

1. Types and safety
   - Types values and operations
   - Typing judgment and inference rules
   - Type safety
   - Type verification for FUN
   - Polymorphism
   - Type inference

# Approach

Writing a type checker for simple types in FUN was relatively easy

- typing rules were syntax-directed,
- some type annotations were required at the few places where we did not have a simple way of guessing the right type.

In the Hindley-Milner system

- the two rules for instantiation and generalization may be applied to any expression.
- we aim at full inference, without any annotation.

# Syntax-directed Hindley-Milner system

We restrict the place where generalization and instantiation may be applied. Type schemes appear only in the environment.

- We allow the instantiation of a type variable only when recovering from the environment the type scheme associated to a variable.
  1. fetch the type scheme $\sigma$ associated to $x$ in $\Gamma$,
  2. instiantiate <u>all</u> universal variables of $\sigma$
- Symmetrically, we allow generalization only for `let` definitions.
  1. type the expression $e_1$ in the environment $\Gamma$, and call $\tau_1$ the obtained type,
  2. generalize <u>all</u> the free variables of $\tau_1$ that can possibly be generalized, to obtain a type schema $\sigma_1$,
  3. type $e_2$ in the extended environment where $x$ is associated to $\sigma_1$.

This syntax-directed variant of the Hindley-Milner system is equivalent to the original version.

# Example

$\Gamma_1 = \{\, x : \alpha \to \alpha, y : \alpha \,\}$ and $\Gamma_2 = \{\, f : \forall \alpha.(\alpha \to \alpha) \to (\alpha \to \alpha) \,\}$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \qquad \dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \qquad \dfrac{}{\Gamma_1 \vdash y : \alpha}}{\Gamma_1 \vdash x\, y : \alpha}}{x : \alpha \to \alpha, y : \alpha \vdash x\,(x\, y) : \alpha}}{x : \alpha \to \alpha \vdash \texttt{fun } y \texttt{ -> } x\,(x\, y) : \alpha \to \alpha}}{\vdash \texttt{fun } x \texttt{ -> fun } y \texttt{ -> } x\,(x\, y) : (\alpha \to \alpha) \to (\alpha \to \alpha)} \qquad (*)}{\vdash \texttt{let } f = \texttt{fun } x \texttt{ -> fun } y \texttt{ -> } x\,(x\, y) \texttt{ in } f\,(\texttt{fun } z \texttt{ -> } z\texttt{+}1) : \texttt{int} \to \texttt{int}}
$$

$$
(*)\dfrac{\dfrac{}{\Gamma_2 \vdash f : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma_2, z : \texttt{int} \vdash z : \texttt{int}} \qquad \dfrac{}{\Gamma_2, z : \texttt{int} \vdash 1 : \texttt{int}}}{\Gamma_2, z : \texttt{int} \vdash z\texttt{+}1 : \texttt{int}}}{\Gamma_2 \vdash \texttt{fun } z \texttt{ -> } z\texttt{+}1 : \texttt{int} \to \texttt{int}}}{f : \forall \alpha.(\alpha \to \alpha) \to (\alpha \to \alpha) \vdash f\,(\texttt{fun } z \texttt{ -> } z\texttt{+}1) : \texttt{int} \to \texttt{int}}
$$

# Constraint generation and unification

The algorithm W implements type inference :

- Each time we need a type which cannot be computed directly, we introduce instead a new type variable.
  - type of the parameter of a function
  - types used for instantiating the universal variables of a type scheme $\Gamma(x)$.
- The actual types represented by these type variables are computed later, when checking/solving the constraints related to the typing rules (for instance, for application or addition).
  - When a typing rule requires an identity between two types $\tau_1$ and $\tau_2$ containing type variables $\alpha_1$, ..., $\alpha_n$, we try to unify these two types, that is we look for an instantiation $f$ of the type variables $\alpha_i$ such that $f(\tau_1) = f(\tau_2)$.

# Examples of unification

- If $\tau_1 = \alpha \rightarrow \texttt{int}$ and $\tau_2 = (\texttt{int} \rightarrow \texttt{int}) \rightarrow \beta$, we can unify the types $\tau_1$ and $\tau_2$ using the instantiation $[\alpha \mapsto \texttt{int} \rightarrow \texttt{int}, \beta \mapsto \texttt{int}]$.
- If $\tau_1 = (\alpha \rightarrow \texttt{int}) \rightarrow (\alpha \rightarrow \texttt{int})$ and $\tau_2 = \beta \rightarrow \beta$, we can unify the types $\tau_1$ and $\tau_2$ using the instantiation $[\beta \mapsto \alpha \rightarrow \texttt{int}]$.
- The types $\alpha \rightarrow \texttt{int}$ and $\texttt{int}$ cannot be unified.
- The types $\alpha \rightarrow \texttt{int}$ and $\alpha$ cannot be unified.

Unification criteria :

- $\tau$ is always unified with itself,
- unification of $\tau_1 \rightarrow \tau'_1$ with $\tau_2 \rightarrow \tau'_2$ requires unifying $\tau_1$ with $\tau_2$ and $\tau'_1$ with $\tau'_2$,
- unification of $\tau$ with a variable $\alpha$, when $\alpha$ does not appears in $\tau$, is done by instantiating $\alpha$ by $\tau$ (if $\alpha$ appears in $\tau$, unification is not possible),
- in any other case, unification is not possible.

# Algorithm W, example

Let us infer a type for the expression
`let` $f$ = `fun` $x$ `->` `fun` $y$ `->` $x(xy)$ `in` $f$ (`fun` $z$`->`$z$+1).
We first focus on `fun` $x$ `->` `fun` $y$ `->` $x$ ($x$ $y$),

- The variable $x$ is given the type $\alpha$, where $\alpha$ is a new type variable.
- Similarly, the variable $y$ is given the type $\beta$ with $\beta$ a new type variable.
- Then we type the expression $x$ ($x$ $y$).
  - The application $x$ $y$ requires the type $\alpha$ of $x$ to be a functional type, whose parameter corresponds to the type $\beta$ of $y$. Thus we unify $\alpha$ with $\beta \to \gamma$, for $\gamma$ some new type variable, and define a first element of instantiation : $\alpha = \beta \to \gamma$.
  - Therefore, the application $x$ $y$ has the type $\gamma$.
  - The application $x$ ($x$ $y$) requires the type $\alpha = \beta \to \gamma$ of $x$ to be a functional type, whose parameter corresponds to the type $\gamma$ of $x$ $y$. Thus we unify $\beta \to \gamma$ with $\gamma \to \delta$, for $\delta$ a new type variable. Then we get new instantiation information : $\gamma = \delta = \beta$.

  We also deduce that the application $x$ ($x$ $y$) has the type $\beta$.
- Finally, `fun` $x$ `->` `fun` $y$ `->` $x$ ($x$ $y$) get the type $\alpha \to (\beta \to \beta)$, which is $(\beta \to \beta) \to (\beta \to \beta)$,
- in the empty typing context this can be generalized as $\forall\beta.(\beta \to \beta) \to (\beta \to \beta)$.

# Algorithm W, example (continued)

We typecheck $f$ (fun $z$ -> $z$+1), in a context where $f : \forall\beta.(\beta \to \beta) \to (\beta \to \beta)$.

- We fetch $f : \forall\beta.(\beta \to \beta) \to (\beta \to \beta)$ from the context, and instantiate the universal variable $\beta$ with a new type variable $\zeta$.
  We get for $f$ the type $(\zeta \to \zeta) \to (\zeta \to \zeta)$.
- Typing of fun $z$ -> $z$+1.
  - The variable $z$ is given the type $\eta$ (new type variable).
  - We type the addition $z$+1.
    - $z$ has the type $\eta$, that has to be unified with int. So $\eta =$ int.
    - 1 has the type int, that has to be unified with int

    Thus $z$+1 has the type int.

  Thus fun $z$ -> $z$+1 has the type $\eta \to$ int, which is int $\to$ int.
- To type the application itself, we have to unify the type $(\zeta \to \zeta) \to (\zeta \to \zeta)$ of $f$ with the type (int $\to$ int) $\to \theta$ of a function that takes a parameter of type int $\to$ int with $\theta$ a new type variable.
  Thus we complete the instantiation with $\zeta =$ int and $\theta =$ int $\to$ int.

Finally, $f$ (fun $z$ -> $z$+1) has the type $\theta =$ int $\to$ int,

$\vdash$ let $f$ = fun $x$ -> fun $y$ -> $x$ ($x$ $y$) in $f$ (fun $z$ -> $z$+1) : int $\to$ int

# Algorithm W, in caml

We take the raw abstract syntax of FUN, without type annotations.

```
type bop = Add | Sub | Mul | Lt | Eq
type expr =
  | Int of int
  | Bop of bop * expr * expr
  | Var of string
  | Let of string * expr * expr
  | If  of expr * expr * expr
  | App of expr * expr
  | Fun of string * expr
  | Fix of string * expr
```

# Algorithm W, in caml

We extend simple types with a notion of type variable, and define a type scheme as a pair of a simple type typ and a set vars of universally quantified type variables.

```
type typ =
  | TInt
  | TBool
  | TFun of typ * typ
  | TVar of string

module VSet = Set.Make(String)
type schema = { vars: VSet.t; typ: typ }
let scheme_of_typ t = { vars= VSet.empty; typ: t }
```

A typing environment associate a type scheme to each variable of the program.

```
module SMap = Map.Make(String)
type env = schema SMap.t
```

# Algorithm W, in caml

We build a function type_inference: expr –> typ that computes a type for the expression given as parameter, trying to get a type that is as general as possible.
This function uses an auxiliary function new_var: unit –> string for creating new type variables.

```
let type_inference t =
  let new_var =
    let cpt = ref 0 in
    fun () -> incr cpt; Printf.sprintf "tvar_%i" !cpt
  in
```

# Representing substitutions

- Type variables are associated to concrete types (or at least more precise types) when new constraints are discovered and analyzed.
- These associations are recorded in a hash table `subst`, that grows as the inference proceeds.

  ```
  let subst = Hashtbl.create 32 in
  ```

  It allows sharing and avoid the cost of substitution
- Thus, the types used during inference will contain type variables, some of which will have a definition in subst.
- To read such a type, we use auxiliary unfolding functions unfold and unfold_full , which take a type $\tau$ and replace its type variables by their definition in subst (for those that have one).

# Unfolding functions

- unfold is a “shallow” replacement : it replaces only what is necessary to distinguish between the cases TInt, TBool, TFun or TVar.
- The function unfold_full performs a complete replacement

```
let rec unfold t = match t with
    | TInt | TBool | TFun _ -> t
    | TVar a ->
        if Hashtbl.mem subst a then
          unfold (Hashtbl.find subst a)
        else
          t
  in

let rec unfold_full t = match unfold t with
    | TFun(t1, t2) -> TFun(unfold_full t1, unfold_full t2)
    | t -> t
  in
```

# Example using unfold

Check whether a type variable $\alpha$ appears in a type $\tau$

```
let rec occur a t = match unfold t with
  | TInt | TBool -> false
  | TVar b -> a=b
  | TFun(t1, t2) -> occur a t1 || occur a t2
```

# Core of the W algorithm

Consistency check is performed by an auxiliary function unify, which records on the fly the new associations between type variables and concrete types.
Auxilliary functions :

- instantiate : schema –> typ replaces each universal variable by a fresh type variable.
- generalize : typ –> env –> schema, returns a type scheme in which type variables not free in the environment are generalized
- unify : typ –> typ –> unit, try to unify its two arguments, adding the constraints in the subst table, otherwise fails

# Core of the W algorithm

```
let rec w e env = match e with
    | Int _ -> TInt
    | Bop((Add | Sub | Mul), e1, e2) ->
       let t1 = w e1 env in
       let t2 = w e2 env in
       unify t1 TInt; unify t2 TInt;
       TInt
    | Bop(Lt, e1, e2) ->
       let t1 = w e1 env in
       let t2 = w e2 env in
       unify t1 TInt; unify t2 TInt;
       TBool
    | Bop(Eq, e1, e2) ->
       let t1 = w e1 env in
       let t2 = w e2 env in
       unify t1 t2;
       TBool
```

# Core of the W algorithm (continued)

```
| If(c, e1, e2) ->
   let tc = w c env in
   let t1 = w e1 env in
   let t2 = w e2 env in
   unify tc TBool;
   unify t1 t2;
   t1
| Var x -> instantiate (SMap.find x env)
| Let(x, e1, e2) ->
   let t1 = w e1 env in
   let st1 = generalize t1 env in
   let env' = SMap.add x st1 env in
   w e2 env'
| Fun(x, e) ->
   let v = new_var() in
   let env = SMap.add x (scheme_of_typ (TVar v)) env in
   let t = w e env in
   TFun(TVar v, t)
```

# Core of the W algorithm (continued)

```
| App(e1, e2) ->
   let t1 = w e1 env in
   let t2 = w e2 env in
   let v = TVar (new_var()) in
   unify t1 (TFun(t2, v));
   v
| Fix(f, e) ->
   let v = new_var() in
   let env = SMap.add f (scheme_of_typ (TVar v)) env in
   let t = w e env in
   unify t (TVar v);
   t
```

# Unification

```ocaml
let rec unify t1 t2 = match unfold t1, unfold t2 with
    | TInt, TInt   -> ()
    | TBool, TBool -> ()
    | TFun(t1, t1'), TFun(t2, t2')
                   -> unify t1 t2; unify t1' t2'
    | TVar a, TVar b when a=b -> ()
    | TVar a, t | t, TVar a ->
       if occur a t then
         failwith "unification error"
       else
         Hashtbl.add subst a t
    | _, _ -> failwith "unification error"
```

# Instantiation

```
let instantiate s =
   let renaming = VSet.fold
        (fun v r -> SMap.add v (TVar(new_var())) r)
        s.vars SMap.empty
   in
   let rec rename t = match unfold t with
    | TVar a as t ->
       (try SMap.find a renaming with Not_found -> t)
    | (TInt | TBool as t) -> t
    | TFun(t1, t2) -> TFun(rename t1, rename t2)
    in rename s.typ
```

# Generalization

```
let rec fvars t = match unfold t with
    | TInt | TBool -> VSet.empty
    | TFun(t1, t2) -> VSet.union (fvars t1) (fvars t2)
    | TVar x -> VSet.singleton x
  in
  let rec schema_fvars s =
    VSet.diff (fvars s.typ) s.vars
  in
  let generalize t env =
    let fvt = fvars t in
    let fvenv = SMap.fold
        (fun _ s vs -> VSet.union (schema_fvars s) vs)
        env
        VSet.empty
    in
    {vars = VSet.diff fvt fvenv; typ=t}
```

# Summary

- A notion of simple types (base types + function types)
- Typing environments, typing judgments
- Inference rules caracterizing “well-typed” expressions (including consistency)
- Proof that well-typed expression can be evaluated “safely”
- Algorithms for type-checking and type inference