Programming Language Semantics and Compiler Design

(Sémantique des Langages de Programmation et Compilation)

Types and Type Analysis

Frédéric Lang & Laurent Mounier
firstname.lastname@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, Inria,
Laboratoire d'Informatique de Grenoble & Verimag

Master of Sciences in Informatics at Grenoble (MoSIG)
Master 1 info

Univ. Grenoble Alpes - UFR IM²AG
www.univ-grenoble-alpes.fr — im2ag.univ-grenoble-alpes.fr

Outline - Types and Type Analysis

Outline: Types and Type Analysis

About Types

What is a type?

- It defines the set of values an expression can take at run-time.
- It defines the set of operations that can be applied to an identifier.
- It defines the resulting type of an expression after applying an operation.

Objectives:

- prevent runtime errors;
- anticipate the runtime behavior.

Example 1 (Types)

int, float, unsigned int, signed int, string, array, list, ...

What are Types Useful for?

Example 2 (Program readability)

```
var e : energy := ... ; -- partition over the variables
var m : mass := ... ;
var v : speed := ... ;
e := 0.5 * (m*v*v) ;
```

Example 3 (Program correctness)

```
var x : kilometers ;
var y : miles ;
x := x + y ; -- typing error
```

Example 4 (Program optimization)

```
var x1, y1, z1 : integer ;
var x2, y2, z2 : real ;
x1 := y1 + z1 ;  -- not the same representations and operators
x2 := y2 + z2 ;
```


Typed and Untyped Languages

Definition 1 (Typed languages)

A dedicated type is associated to each identifier
(and hence to each expression).

Example 5 (Typed languages)

C, C++, Java, Ada, C, CAML, Rust, etc.
What about Python ? JavaScript ?? PhP ???

Remark strongly typed vs weakly typed languages... □

Definition 2 (Untyped languages)

A single (universal) type is associated to each identifier
(and hence to each expression).

Example 6 (Untyped languages)

Assembly language, shell-script, Lisp, etc.

Definition 3 (Explicitly vs implicitly typed languages)

Explicitly typed when types are part of the syntax (implicitly typed otherwise).

Type Safety

"Well-typed programs never go wrong. . . "

(Robin Milner)

Trapped errors vs untrapped errors.

- trapped errors cause computation to stop immediately (e.g., division by 0, access to an illegal address);
- untrapped errors may go unnoticed (e.g., access to a non valid address).

Type safety

A well-typed program never executes "out-of-semantics" behaviors

⇒ **NO** meaningless well-typed programs
(no untrapped errors, no undefined behaviors)

Example 7

Type safe languages

- C, C++ are **(definitely!) not** type safe
- ML, Rust are type safe
- Java, C#, Python, OCaml are "considered as" type safe

Types and type constructions

Basic types

integers, boolean, characters, etc.

Type constructions

- cartesian product (tuples, structures)
- disjoint union
- arrays
- functions
- pointers
- recursive types
- . . .

But also:
subtyping, polymorphism, overloading, inheritance, coercion, overriding, etc.
[see http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf]

Subtyping

Subtyping is a preorder relation $\leq_T$ between types.

It defines a notion of substitutability:

If $T_1 \leq_T T_2$,
then elements of type T_2 may be replaced with elements of type T_1.

Example 8 (Sub-typing)

- class inheritance in OO languages ;
- Integer $\leq_T$ Real (in several languages) ;
- Ada :

```
type Month is Integer range 1..12 ;
-- Month is a subtype of Integer
```


Univ. Grenoble Alpes, UFR IM^{2}AG, MoSIG 1 PLCD - Master 1 info SLPC Types and Type Analysis

Type Checking vs Type inference

In a typed language, the set of "correct typing rules" is called the type system.

The static semantic analysis phase uses this type system in two ways:

Type checking

Check whether "type annotations" are used in a consistent way throughout the program.

Type inference

Compute a consistent type for each program fragment.

Remark In some languages (e.g., Haskel, CAML), there are/can be no type annotations at all (all types are/can be infered). □

Static checking vs dynamic checking

Static checking

Verification performed at compile-time.

Dynamic checking

Verification performed at run-time.
→ necessary to correctly handle:

- dynamic binding for variables or procedures
- polymorphism
- array bounds
- subtyping
- etc.

⇒ For most programming languages, both kinds of checks are used...

Outline: Types and Type Analysis

Getting the Intuition on Examples

- "2 + 3 = 6" is well-typed
- "2 + true = false" is not well-typed
- "x = false" is well-typed
  if x is a (visible) Boolean variable
- "2 + x = y" is well-typed
  if x and y are (visible) integer/real variables
- "let x = 3 in x + y" is well-typed
  if y is a (visible) integer/real variable

$\Rightarrow$ a term t can be type-checked
under assumptions about its **free variables** ...

How to Formalize a Type System?

- Abstract syntax describes terms (represented by ASTs).
- Environment Γ: *Name* $\stackrel{\text{part.}}{\to}$ **Type**.
- Judgment $\Gamma \vdash t : \tau$.
  "In environment Γ*, term t is well-typed and has type* τ*."*
  (free variables of t belong to the domain of Γ)
- Type system

Inference rules	**Axioms**
$\dfrac{\Gamma_1 \vdash \mathcal{A}_1 \quad \cdots \quad \Gamma_n \vdash \mathcal{A}_n}{\Gamma \vdash \mathcal{A}}$	$\Gamma \vdash \mathcal{A}$

Remark A type system is an inference system. □

Example: natural numbers

$e \quad := \quad n \mid x \mid e_1 + e_2$	Syntax
$\dfrac{\Gamma(x) = \textbf{Nat}}{\Gamma \vdash x : \textbf{Nat}}$	x is of type **Nat** in environment Γ if $\Gamma(x) = \textbf{Nat}$.
$\dfrac{}{\Gamma \vdash n : \textbf{Nat}}$	The denotation n is of type **Nat**.
$\dfrac{\Gamma \vdash e_1 : \textbf{Nat} \quad \Gamma \vdash e_2 : \textbf{Nat}}{\Gamma \vdash e_1 + e_2 : \textbf{Nat}}$	$e_1 + e_2$ is of type **Nat** assuming that e_1 and e_2 are of type **Nat**.

Derivations in a Type System

A type-check is a proof in the type system, i.e., a derivation tree where:

- leaves are axioms,
- nodes are obtained by application of inference rules.

A judgment is valid iff it is the root of a derivation tree.

Example 9

$$\dfrac{\emptyset \vdash 1 : \textbf{Nat} \qquad \emptyset \vdash 2 : \textbf{Nat}}{\emptyset \vdash 1 + 2 : \textbf{Nat}}$$

Exercise

Prove that $[x \to \textbf{Nat}, y \to \textbf{Nat}] \vdash x + 2 : \textbf{Nat}$.

Outline: Types and Type Analysis

Outline: Types and Type Analysis

Syntax of Language **While**

Expressions

Expressions: informal description

- same syntax for Boolean and integer expressions (e).
- 3 kinds of (syntactically) distinct binary operators: arithmetic (opa), boolean (opb) and relational (`oprel`)
- (The following can be easily extended to account for unary operators over integers and booleans.)

Expressions: abstract grammar

$$\text{e} \quad ::= \quad \texttt{true} \mid \texttt{false} \mid \text{n} \mid \text{x} \mid \text{e opa e} \mid \text{e oprel e} \mid \text{e opb e}$$

where `true` and `false` are the boolean constants, `n` denotes a natural number, and `x` denotes a variable.

Expressions

Let us consider opa to be $\{+, -, \times, \ldots\}$, opb to be $\{\texttt{and}, \texttt{or}, \ldots\}$, and `oprel` to be { =, <, >, $\neq$, ... }. For example:

- `42`
- `x1`
- `2 × x2`
- `x1 = x2`
- `false`
- `x1 + 42`
- `xb and x2>0`
- `x1 > x2`

are expressions obtained with the above grammar.

Syntax of Language **While**

Statements

S	::=	x := e	(assignment of an expression to a variable x)
		| skip	(doing nothing)
		| S ; S	(sequential composition)
		| if e then S else S fi	(conditional composition)
		| while e do S od	(iterative and unbounded composition)

Statements

Assume a set of variable names `x, x1, y, z`. For example:

```
skip
```

```
x1 := 32
```

```
x1 := 42;
 y := x1 + 12
```

```
...
x1 := 42 + y;
if (x1 + z > 0) then
   x1 := 0;
    y := 42
else
   x1 := 42;
    y := 0
fi
```

```
...
while (x > 0) do
   y := y + z;
   x := x - 1
od
```

are statements obtained with the above grammar.

Judgments

- $\Gamma \vdash S$
  "In environment Γ, statement S is well-typed".

- $\Gamma \vdash e : t$
  "In environment Γ, expression e is of type t".

Type System for Expressions

bool. constant	int. constant	int opbin
$\dfrac{}{\Gamma \vdash \texttt{true} : \textbf{Bool}}$ $\dfrac{}{\Gamma \vdash \texttt{false} : \textbf{Bool}}$	$\dfrac{}{\Gamma \vdash \texttt{n} : \textbf{Int}}$	$\dfrac{\Gamma \vdash e_1 : \textbf{Int} \quad \Gamma \vdash e_2 : \textbf{Int}}{\Gamma \vdash e_1 \ \texttt{opa} \ e_2 : \textbf{Int}}$

variables	bool. opbin	relational operators
$\dfrac{\Gamma(x) = t}{\Gamma \vdash x : t}$	$\dfrac{\Gamma \vdash e_1 : \textbf{Bool} \quad \Gamma \vdash e_2 : \textbf{Bool}}{\Gamma \vdash e_1 \ \texttt{opb} \ e_2 : \textbf{Bool}}$	$\dfrac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \ \texttt{oprel} \ e_2 : \textbf{Bool}}$

Type system for Statements

Assignment	**Skip**
$\dfrac{\Gamma \vdash e : t \quad \Gamma \vdash x : t}{\Gamma \vdash x := e}$	$\dfrac{}{\Gamma \vdash \texttt{skip}}$

Sequence	**Iteration**
$\dfrac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1 ; S_2}$	$\dfrac{\Gamma \vdash e : \textbf{Bool} \quad \Gamma \vdash S}{\Gamma \vdash \text{while } e \text{ do } S \text{ od}}$

Exercises

Exercise: conditional statement

Complete the type system by providing a rule for conditional statements.

Exercise: unary operators

Complete the abstract syntax and the type system by providing rules for unary operators.

Exercise: introducing reals and type conversion

Extend the type system for the expressions assuming that arithmetic types can be now either integer (**Int**) or real (**Real**).

Several solutions are possible:

1. Type conversions are never allowed.
2. Only explicit conversions (with a cast operator) are allowed.
3. (implicit) conversions are allowed.

Programs

Definition 4 (Program abstract syntax)

The syntactic category **Prog** of programs $(C, C', C_0, \ldots)$ is defined as follows:

$$C \quad ::= \quad \text{global } D_V \text{ in } S$$
$$D_V \quad ::= \quad x : t \;,\; D_v \mid \varepsilon$$

where:

- $D_v = x_1 : t_1, \ldots, x_n : t_n \quad (n \geq 0)$ are the global variables, where: $x_i : t_i \in$ **Var** $\times$ **Type** declares a variable x_i **of type** t_i (see later for other forms of variable declarations).
- S is the program body.

Example 10 (Program)

The following program initializes x and y and then swaps their values:

```
global x:Int, y:Int, t:Int in
   x:=12; y:=42 ;
   t:=x ; x:=y ; y:=t
```


Extending the judgements

Notations

- **vars**(D_V) denotes the set of variables **declared** in D_V.
- $\Gamma[y \mapsto \tau]$ denotes the environment Γ' such that:
  - $\Gamma'(x) = \Gamma(x)$ if $x \neq y$
  - $\Gamma'(y) = \tau$

Judgments

- $\Gamma \vdash \; D_V \mid \Gamma_l$ means
  "declarations D_V update environment Γ into Γ_l"
- $\Gamma \vdash \; S$ means
  "statement S is well-typed within environment Γ"
- $\Gamma \vdash \; C$ means
  "program C is well-typed within environment Γ"

Extending the Type System

Inference rule for programs

$$\frac{\emptyset \vdash D_V \mid \Gamma_g \quad \Gamma_g \vdash S}{\emptyset \vdash \text{global } D_V \text{ in } S}$$

Inference rules for declarations

$$\frac{}{\Gamma \vdash \epsilon \mid \Gamma} \qquad \frac{\Gamma[x \mapsto t] \vdash D_V \mid \Gamma' \quad x \notin \textbf{vars}(D_V)}{\Gamma \vdash \textbf{x : t} \,;\, D_V \mid \Gamma'}$$

$x \notin \textbf{vars}(D_V)$ prevents double variable declarations

Exercise: applying the typing rules

1. check that `global x:Int, y:Int in x:=1 ; y:=x+1` is well-typed.
2. check that `global x:Int, y:Int in x:=1 ; y:=z+1` is not well-typed.

Some Alternatives for Variable Declarations

Inference rules for declarations allowing re-declarations

$$\frac{}{\Gamma \vdash \epsilon \mid \Gamma} \qquad \frac{\Gamma[x \mapsto t] \vdash D_V \mid \Gamma_l}{\Gamma \vdash \; x : t \,;\, D_V \mid \Gamma_l[x \mapsto t]}$$

gives priority to the **last** declarations.

Alternative declarations

- initialized variables: `x:=e:t`
- untyped initialized variables: `x:=e`
- uninitialized and untyped variables: `x`

We will study these alternatives during the tutorial ...

Outline: Types and Type Analysis

(Parameterless) Procedure definitions

Definition 5 (Syntactic categories of **Proc**)

- **Pid**: procedure identifiers (written $F, F_0, F_1, \ldots$)
- **Pdef**: procedure definitions (written $D, D_0, D_1, \ldots$)
- **Prog**: programs (written $C, C_0, C_1, \ldots$)

Definition 6 (Syntax of procedure definitions)

The syntax of procedure definitions $D_P \in$ **Pdef** is as follows:

$$D_P ::= \text{proc } F \text{ is var } D_V \text{ in } S \text{ end}$$
$$D_V ::= x : t, D_V \mid \varepsilon$$

where:

- $F \in$ **Pid** is the procedure identifier.
- $S \in$ **Stm** is the procedure body.
- $D_V = x_1 : t_1, \ldots, x_n : t_n \in$ **Var** $\times$ $(n \geq 0)$ are the local variable declarations. If $n = 0$ then the keyword var may be omitted.

(Procedure parameters will be introduced later)

Programs with (parameterless) procedures

Definition 7 (Program abstract syntax)

The syntactic category **Prog** of programs $(C, C', C_0, \ldots)$ is defined as follows:

$$\begin{array}{rcl} C & ::= & \text{global } D_V \ D_{Proc} \text{ in } S \\ D_V & ::= & x : t,\ D_v \mid \varepsilon \\ D_{Proc} & ::= & D_P\ , D_{Proc} \mid \varepsilon \end{array}$$

where:

- $D_V = x_1 : t_1, \ldots, x_n : t_n \ \ (n \geq 0)$ are the global variable declarations
- $D_{Proc} = D_1, \ldots, D_m \in$ **Pdef** $(m \geq 0)$ are the procedure declarations.
- S is the program body.

Example 11 (Program)

The following program initializes x and y and then swaps their values:

```
global x:Int, y:Int in
   proc swap_x_y var t:Int in t:= x; x:= y; y:=t end
   x:=12; y:=42 ;
   swap_x_y
```


Notations and Judgments

- A Procedure environment is a (partial) function $\Gamma_P : \textbf{Pid} \rightarrow \{proc\}$
- **procs**(D_P) denotes the set of procedures **declared** in D_P.
- For variable environments Γ_1 and Γ_2, $\Gamma_1[\Gamma_2]$ denotes the environment Γ' such that:
  - $\Gamma'(x) = \Gamma_2(x)$ if $x \in Dom(\Gamma_2)$
  - $\Gamma'(x) = \Gamma_1(x)$ if $x \in Dom(\Gamma_1) \setminus Dom(\Gamma_2)$
- $\Gamma_V \vdash \ D_V \mid \Gamma'_V$ means

  Variable declarations D_V update variable environment Γ_V into Γ'_V".
- $(\Gamma_V, \Gamma_P) \vdash D_P \mid \Gamma'_P$ means

  "Procedure declarations in D_P are well-typed within variable environment Γ_V and procedure *environments Γ_P. Moreover, procedure declarations in D_P update procedure environment Γ_P into Γ'_P."*
- $(\Gamma_V, \Gamma_P) \vdash S$ means

  "Statement S is well-typed within variable environment Γ_V and procedure *environments Γ_P.*

Extending the Type System

Inference rule for programs

$$\frac{\emptyset \vdash D_V \mid \Gamma_g \quad (\Gamma_g, \emptyset) \vdash D_P \mid \Gamma_P \quad (\Gamma_g, \Gamma_P) \vdash S}{\emptyset \vdash \text{global } D_V \; D_P \text{ in } S}$$

Inference rules for procedure declarations

$$\frac{}{(\Gamma_g, \Gamma_P) \vdash \epsilon \mid \Gamma_P}$$

$$\frac{\emptyset \vdash D_V \mid \Gamma_l \quad (\Gamma_V, \Gamma_P) \vdash S \quad (\Gamma_g, \Gamma_P[F \mapsto proc]) \vdash D_P \mid \Gamma'_P \quad F \notin \textbf{procs}(D_P)}{(\Gamma_g, \Gamma_P) \vdash \text{ proc } F \text{ is var } D_V \text{ in } S \text{ end } ; D_P \mid \Gamma'_P}$$

with $\Gamma_V = \Gamma_g[\Gamma_l]$, where local variables hide global ones ...

Inference rule for procedure calls

$$\frac{F \in Dom(\Gamma_P)}{(\Gamma_V, \Gamma_P) \vdash F}$$

Exercise: playing with these inference rules ...

Type check the following programs:

C1

```
global x:Int, y:Int in
   proc swap_x_y var t:Int in t:= x; x:= y; y:=t end
   x:=12; y:=42 ;
   swap_x_y
```

C2

```
global x:Int, y:Int, t:int in
   proc swap_x_y var x:Int, y:Int in t:= x; x:= y; y:=t end
   x:=12; y:=42 ;
   swap_x_y
```

C3

```
global x:Bool, y;Bool  in
   proc swap_x_y var x:int, y:Int, t:Int in t:= x; x:= y; y:=t end
   x:=true; y:=true ;
   swap_x_y
```

Procedures with parameters

Definition 8 (Updated Syntax of procedure definitions)

The syntax of procedure definitions $D_P \in$ **Pdef** is as follows:

$$D_P \quad ::= \quad \text{proc } F(D_F) \text{ is var } D_V \text{ in } S \text{ end}$$

where:

$D_F = y_1 : t_1, \ldots, y_n : t_n \in \textbf{Var} \times \textbf{Type}(n \geq 0)$ is the formal parameters declaration.

(if $n = 0$ then the parenthesis may be omitted)

Definition 9 (Updated Syntax of procedure calls)

$$S \quad ::= \quad \cdots \mid F(e_1, \ldots e_n)$$

where **expressions** e_i are the actual parameters

Rk: we consider here value-passing parameters
(without distinguishing between input/output modes).

Example 12 (Procedure declaration)

```
proc max_z(a:Int, b:Int) is var m:Int in
    if (a>b) then m:=a else m:=b fi ; z:=m end
```


Programs with procedures

Definition 10 (Program abstract syntax (unchanged))

The syntactic category **Prog** of programs $(C, C', C_0, \ldots)$ is defined as follows:

$$\begin{array}{rcl} C & ::= & \text{global } D_V \ D_{Proc} \text{ in } S \\ D_{Proc} & ::= & D_P, \ D_{Proc} \mid \varepsilon \\ D_V & ::= & x : t, \ D_v \mid \varepsilon \end{array}$$

where:

- $D_V = x_1 : t_1, \ldots, x_n : t_n \ \ (n \geq 0)$ are the global variable declarations
- $D_{Proc} = D_1, \ldots, D_m \in$ **Pdef** $(m \geq 0)$ are the procedure declarations.
- S is the program body.

Example 13 (Program)

```
global x:Int, y:Int, z;Int in
proc max_z(a:Int, b:Int) is var m:Int in
    if (a>b) then m:=a else m:=b fi ; z:=m end
x:=12; y:=42 ; max_z(x*4,y-1)
```

Collisions between identifier names?

What to decide when a same identifier is used for a global variable, a formal parameter and a local variable ?

A syntactically correct program

```
global x:Int in
proc foo(x:Bool) is var x:Real in x:= ... end
...
```

From a type-checking viewpoint:

1. Is this program considered as correct?
2. What could be a valid right-hand side for the assigment of `x` within `foo`?

Proposed answers (compatible with NOS!):

possible collisions betwen global and local idfs, priority to local ones

possible collisions betwen variable and parameter idfs, priority to parameters.

Rk: collisions between procedure and variable/parameter idfs are rejected

Extending (once again !) the Type System

A Procedure environment is now a (partial) function $\Gamma_P : \mathbf{Pid} \to (\mathbf{Type})^*$ mapping each procedure name to the sequence of its **formal parameter types**. For a formal parameter declaration $D_F = (y_1 : t_1, \ldots y_n : t_n)$ we note $\mathbf{paramtypes}(D_F)$ the sequence $(t_1, \ldots t_n)$.

Inference rules for procedure declarations

$$\frac{}{(\Gamma_g, \Gamma_P) \vdash \epsilon \mid \Gamma_P}$$

$$\frac{\emptyset \vdash D_F \mid \Gamma_f \quad \emptyset \vdash D_V \mid \Gamma_l \quad (\Gamma_V, \Gamma_P) \vdash S \quad (\Gamma_g, \Gamma_P[F \mapsto t_F]) \vdash D_P \mid \Gamma'_P \quad F \notin \mathbf{procs}(D_P)}{(\Gamma_g, \Gamma_P) \vdash \text{ proc } F(D_F) \text{ is var } D_V \text{ in } S \text{ end ; } D_P \mid \Gamma'_P}$$

where $t_F = \mathbf{paramtypes}(D_F)$

where $\Gamma_V = (\Gamma_g[\Gamma_l])[\Gamma_f]$, meaning that parameters hide local/global variables ...

Inference rule for procedure calls

$$\frac{\Gamma_P(F) = (t_1, \ldots, t_n) \quad \Gamma_V \vdash e_i : t_i}{(\Gamma_V, \Gamma_P) \vdash F(e_1, \ldots, e_n)}$$

Rk: types of formal and actual parameters should agree position-wise

Exercises

Applying the rules

Type check the two following programs:

```
global r:Int, x:Int in
  proc add_r(x:Int, y;Int) is var s:Int in s:=x+y ; r:=s end
x:=5; add_r(x,x+2)
```

```
global r:Int in
  proc foo (x:Int) is r:=x+1 ; foo(x)
r:=5; foo(r)
```

Extensions and variants

- recursive procedure calls
- mutally recursive procedure calls
- collateral evaluation of procedure declarations
- input/output procedure parameters
- functions (extending the procedure declaration syntax)

↪ Some of these extensions will be studied during the tutorials ...

Outline: Types and Type Analysis

A Small Functional Language

Syntax of the language

$$\begin{array}{lcl} e & ::= & n \mid r \mid \textbf{true} \mid \textbf{false} \mid x \mid \textbf{fun}\ x : \tau . e \mid (e\ e) \mid (e\ ,\ e) \\ \tau & ::= & \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} \mid \tau \rightarrow \tau \mid \tau \times \tau \end{array}$$

Example 14 (Programs)

- 42
- $(x\ 12.5)$
- $(x\ ,\ true)$
- $\textbf{fun}\ x : \textbf{Bool}.\ x$
- $((\textbf{fun}\ x : \textbf{Bool}.\ x)\ 12)$
- $\textbf{fun}\ x : \textbf{Int} \rightarrow \textbf{Real}.\ (x\ 12)$

Version 1: no polymorhism, explicit type annotations

Judgment

$\Gamma \vdash e : \tau$ means "In environment Γ, e is well-typed and of type τ."

Type System

$$\frac{}{\Gamma \vdash n : \textbf{Int}} \quad \frac{}{\Gamma \vdash r : \textbf{Real}} \quad \frac{}{\Gamma \vdash \textbf{true} : \textbf{Bool}} \quad \frac{}{\Gamma \vdash \textbf{false} : \textbf{Bool}}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \textbf{fun}\ x : \tau_1 . e : \tau_1 \mapsto \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ ,\ e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \mapsto \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

Extension: definition of identifiers

We add a new construct:

$$\textbf{let } x = e_1 : \tau_1 \textbf{ in } e_2$$

Informal semantics:

within e_2, each occurrence of x is replaced by e_1

Extending the type system to handle identifiers

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let } x = e_1 : \tau_1 \textbf{ in } e_2 : \tau_2}$$

Version 2: no polymorphism, no type annotations

Syntax of the language

$$e ::= \cdots \mid \textbf{fun } x.e \mid \textbf{let } x = e_1 \textbf{ in } e_2$$

Modified type system

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \textbf{fun } x.e : \tau_1 \mapsto \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2}$$

$\Rightarrow$ a unique value for type τ_1 has to be infered ...

Examples

Expressions that can be typed:

- $((\textbf{fun } x.x)\ 1) : \textbf{Int}$
- $((\textbf{fun } x.x)\ \textbf{true}) : \textbf{Bool}$
- $\textbf{let } x = 1 \textbf{ in } ((\textbf{fun } y.y)\ x) : \textbf{Int}$
- $\textbf{let } f = \textbf{fun } x.x \textbf{ in } (f\ 2) : \textbf{Int}$

Expressions that cannot be typed
$\nexists(\Gamma, \tau)$ such that $\Gamma \vdash e : \tau$

- $(1\ 2)$
- $\textbf{fun } x.(x\ x)$
- $\textbf{let } f = \textbf{fun } x.x \textbf{ in } ((f\ 1)\ ,\ (f\ \textbf{true}))$

Univ. Grenoble Alpes, UFR IM^{2}AG, MoSIG 1 PLCD - Master 1 info SLPC Types and Type Analysis

Polymorphism?

We introduce:

- type variable α
- $\forall\alpha.\tau$ means "α can take any type within type expression τ"

Example 15 (Polymorphic expression)
fun $x.x$ is of type $\forall\alpha.\alpha \rightarrow \alpha$

Definition 11 (Set of free type variables)
Given an environment Γ:

$$\mathcal{D}(\textbf{Bool}) = \mathcal{D}(\textbf{Int}) = \mathcal{D}(\textbf{Real}) = \emptyset$$

$$\begin{array}{rcl} \mathcal{D}(\alpha) & = & \{\alpha\} \\ \mathcal{D}(\tau_1 \longrightarrow \tau_2) & = & \mathcal{D}(\tau_1) \cup \mathcal{D}(\tau_2) \\ \mathcal{D}(\forall\alpha \cdot \tau) & = & \mathcal{D}(\tau) \setminus \{\alpha\} \\ \mathcal{D}(\Gamma) & = & \bigcup_{x \in \textbf{dom}(\Gamma)} \mathcal{D}(\Gamma(x)) \end{array}$$

Polymorphism: the F system

Definition 12 (Rules for system F)

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{D}(\Gamma)}{\Gamma \vdash e : \forall \alpha \cdot \tau} \quad \text{(generalization)}$$

$$\frac{\Gamma \vdash e : \forall \alpha \cdot \tau}{\Gamma \vdash e : \tau[\tau' \mapsto \alpha]} \quad \text{(instanciation)}$$

Example 16 (Programs)

- **let** f = **fun** $x.x$ **in** ((f 1) , (f **true**))
- **fun** $x.(x\ x)$

Remark Type inference is no longer decidable in this type system... □

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC Types and Type Analysis

Polymorphism: the Hindley-Milner system

Type quantifiers may only appear "in front" of type expressions.

Definition 13 (New Syntax)

Types	τ	::=	**Bool** | **Int** | **Real** | $\tau \longrightarrow \tau$ | $\tau \times \tau$ | α
Type patterns	σ	::=	τ | $\forall \alpha \cdot \sigma$.

Definition 14 (New Rules for the Hindley-Milner system)

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{D}(\Gamma)}{\Gamma \vdash e : \forall \alpha \cdot \sigma} \quad \text{(generalization)}$$

$$\frac{\Gamma \vdash e : \forall \alpha \cdot \sigma}{\Gamma \vdash e : \sigma[\tau \mapsto \alpha]} \quad \text{(instanciation)}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma[x \mapsto \sigma_1] \vdash e_2 : \sigma_2}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \sigma_2} \quad \text{(polymorph "let")}$$

Example 17

let f = **fun** $x.x$ **in** ((f 1) , (f **true**))

A “Python-like” Language

Syntax of the language

Similar as **While** language . . . except that variables are no longer declared !

$$S ::= \cdots \mid \textbf{begin} S \textbf{ end}$$

$\hookrightarrow$ the type of a variable is infered each time it is assigned

Example 18 (Programs)

```
begin
  y:=x+1 ;
  x:=y>8 ;
  if x then
     x:=y*2 ;
     y:=true
  else
    y:=y+1
  fi ;
  x:=y*2
 end
```


Informal typing rules

We consider the same typing rules as for **While**

- two basic types **Int** and **Bool**
- no arithmetic operations between booleans
- no boolean operations between integers
- comparisons only between values of the same types
- etc.

Example 19 (Programs)

Is this program well-typed?

```
begin
  y:=x+1 ;
  x:=y>8 ;
  if x then
     x:=y*2 ;
     y:=true
  else
    y:=y+1
  fi ;
  x:=y*2
 end
```

Solution 1 (static typing): judgments

- Expressions are unchanged from **While**

$$\Gamma \vdash e : t$$

"In environment Γ*, expression* e *is of type* t*".*

- Statements may change the environment ...

$$\Gamma \vdash S \mid \Gamma'$$

"Statement S *updates environment* Γ *into* Γ'*"*

Solution 1 (static typing): rules

Assignment	**Skip**
$\dfrac{\Gamma \vdash e : t \quad \cancel{\Gamma \vdash x : t}}{\Gamma \vdash x := e \mid \Gamma[x \to t]}$	$\dfrac{}{\Gamma \vdash \texttt{skip} \mid \Gamma}$

Sequence	**Conditionnal**
$\dfrac{\Gamma \vdash S_1 \mid \Gamma' \quad \Gamma' \vdash S_2 \mid \Gamma''}{\Gamma \vdash S_1; S_2 \mid \Gamma''}$	$\dfrac{\Gamma \vdash e : \textbf{Bool} \quad \Gamma \vdash S_1 \mid \Gamma_1 \quad \Gamma \vdash S_2 \mid \Gamma_2}{\Gamma \vdash \text{if } e \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \Gamma_1 \cap \Gamma_2}$

Solution 1 (static typing): may reject correct programs . . .

Example 20 (Programs)

Is this program well-typed?

```
begin
  y:=1 ;
  x:=y>8 ;
  if x then           # x is always false
     x:=y*2 ;
     y:=true          # y becomes a Bool
  else
    y:=y+1            # y remains an Int
  fi ;
  x:=y*2              # y is no longer defined in Gamma ...
 end
```

The program is statically rejected, although no type error occurs since the "then" branch is never taken at runtime . . .

Solution 2: dynamic typing - Judgments

Clue: performs the type checking at runtime
$\hookrightarrow$ needs to mix dynamic semantic and type checking rules !

- Rules for Expressions and Statements are expressed using Natural Operationnal Semantics (NOS)
- Configurations for Statements are in $(\textbf{Exp} \times \textbf{State} \times \textbf{Env}) \cup (\textbf{Val} \times \textbf{Type})$:
- Configurations for Expressions are in $(\textbf{Stm} \times \textbf{State} \times \textbf{Env}) \cup (\textbf{State} \times \textbf{Env})$:

Example 21

Solution 2: dynamic typing - Rules

Skip and Assignment

$$\frac{}{(\mathsf{skip}, \sigma, \Gamma) \rightarrow (\sigma, \Gamma)}$$

$$\frac{(e, \sigma, \Gamma) \rightarrow (v, t)}{(x := e, \sigma, \Gamma) \rightarrow (\sigma[x \mapsto v], \Gamma[x \mapsto t])}$$

Sequential Composition

$$\frac{(S_1, \sigma, \Gamma) \rightarrow (\sigma', \Gamma') \quad (S_2, \sigma', \Gamma') \rightarrow (\sigma'', \Gamma'')}{(S_1; S_2, \sigma, \Gamma) \rightarrow (\sigma'', \Gamma'')}$$

Solution 2: dynamic typing - Rules (continued)

Conditional Statements

$$\frac{(b, \sigma, \Gamma) \rightarrow (\mathit{true}, \mathit{Bool}) \quad (S_1, \sigma, \Gamma) \rightarrow (\sigma', \Gamma')}{(\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, \sigma, \Gamma) \rightarrow (\sigma', \Gamma')}$$

$$\frac{(b, \sigma, \Gamma) \rightarrow (\mathit{false}, \mathit{Bool}) \quad (S_2, \sigma, \Gamma) \rightarrow (\sigma', \Gamma')}{(\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, \sigma, \Gamma) \rightarrow (\sigma', \Gamma')}$$

Exercise 1

Rule for the iterative statement ?

Beyond "pure" type checking ?

Some "typing rules" may not **stricly** concern *types* ...

Example: un-initialized variables in Java

each variable should be defined (i.e., assigned) before being used (↪ **dataflow (def-use)** *relation)*

```
int x ;
int y ;
y = x+1 ; // ERROR: x is used before being assigned
```

Formalizing this rule with judgements ?

- Environment Γ is the set of def variables

$$\Gamma \subseteq 2^{Name} \quad (x \in \Gamma \text{ iff } x \text{ has been initialized})$$

- Judgment for Statement:

$$\Gamma \vdash S \mid \Gamma'$$

S is "well-typed" within Γ if it uses only variables in Γ and it produces a new environment Γ′

Outline: Types and Type Analysis

Reminder

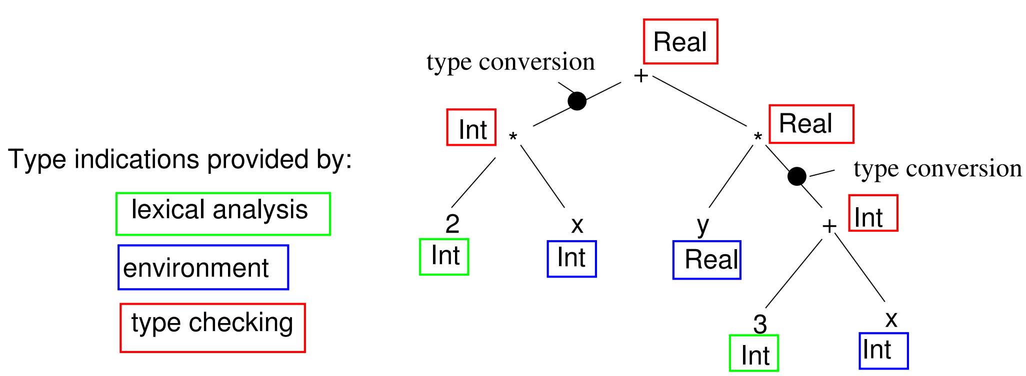
Several issues to be handled during static semantic analysis:

1. type-check the input AST
   - formal specification = a type system
   - notion of environment (name binding), to be computed:
     $\Gamma_V : Name \rightarrow Type$
     $\Gamma_P : Name \rightarrow \{proc\}$
2. decorate this AST to prepare code generation
   - give a type to intermediate nodes
   - indicate implicit type conversions

⇒ How to go from type system to algorithms?

Example

```
begin
   var x : Int ;
   var y : Real ;
   y := 2 * x + y * (3 + x) ;
end
```



Final AST

From a Type System to Algorithms?

⇒ recursive traversal of the AST...

AST representation:

```
typedef struct tnode {
    String string ; // lexical representation
    kind elem ; // category (idf, binaop, while, etc.)
    struct tnode *left, *right ; // children
    Type type ; // type (Int, Real, Void, Bad, etc.)
    ...
} Node ;
```

Type-checking function:

```
Type TypeCheck(* node) ;
// checks the correctness of node, returns the result Type
// and inserts type conversions when necessary
```


Type Checking Algorithm for Arithmetic Expressions

DENOT	BINAOP	IDF
$\dfrac{}{\Gamma \vdash \mathtt{n} : \textbf{Int}}$	$\dfrac{\Gamma \vdash e_l : Tl \quad \Gamma \vdash e_r : Tr \quad T = resType(Tr, Tl)}{\Gamma \vdash e_l \ \mathtt{binaop}\ e_r : T}$	$\dfrac{\Gamma(x) = t}{\Gamma \vdash x : t}$

```
function Type typeCheck(Node *node) {
 switch node->elem {
   case DENOT: break ; // lexical analysis
   case IDF: node->type=Gamma(node->string); break; // environment
   case BINAOP: // type-checking
     Tl=typeCheck(node->left);
     Tr=typeCheck(node->right);
     node->type=resType(Tl, Tr);
     if (node->type != Tl) insConversion(node->left, node->type);
     if (node->type != Tr) insConversion(node->right, node->type);
     break ;
 }
 return node->type ;
}

function Type resType(Type t1, Type t2) {
 if (t1==Boolean) or (t2==Boolean) return Bad; else return Max(t1, t2);
}
```

Type Checking Algorithm for Statements

Sequence	Iteration	Assignment
$\dfrac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1; S_2}$	$\dfrac{\Gamma \vdash e : \textbf{Bool} \quad \Gamma \vdash S}{\Gamma \vdash \texttt{while} \quad e \texttt{ do } S}$	$\dfrac{\Gamma \vdash x : t \quad \Gamma \vdash e : t}{\Gamma \vdash x := e}$

```
function Type typeCheck(Node *node) {
 switch node->elem {
   case SEQUENCE:
     if (typeCheck(node->left) != Void) return BAD ;
     return typeCheck(node->right) ;
   case WHILE:
     if (typeCheck(node->left) != BOOL) return BAD ;
     return typeCheck(node->right) ;
   case ASSIGN:
     Tl=typeCheck(node->left);
     Tr=typeCheck(node->right);
     if (Tl != Tr) return BAD else return VOID ;
  }
}
```


Environment Implementation and Name Binding?

- Associate a type to each identifier
  - each use occurrence $\mapsto$ decl occurrence
  - info should be retrieved efficiently (no AST traversal)
- distinguish between global/local variables, procedure names and formal parameters

Usual Solution: symbol tables

- Store all information associated to an identifier:
  type, kind (var, param, proc), address (for code gen), etc.
- Built during traversals of the declaration parts of the AST
- Efficient search procedure: binary tree, hash table, etc.
- Several solutions for handling nested environment, e.g., $(\Gamma_g[\Gamma_l])[\Gamma_f]$
  - a global table, with a unique qualifying id associated to each idf:

    $$\{((\texttt{x}, \texttt{global}) : \texttt{Int}), ((\texttt{x}, \texttt{local}, \texttt{foo}) : \texttt{Real}), ((\texttt{x}, \texttt{param}, \texttt{foo}) : \texttt{Bool})\}$$

  - a dynamic stack of local tables, one local table per environment, ordered by priority access, and updated at each procedure entry/exit

    $$\{\texttt{x:Int, ...}\} \longrightarrow \{\texttt{x:Real, ...}\} \longrightarrow \{\texttt{x:Bool, ...}\}$$

Outline: Types and Type Analysis

Conclusion

- Types are useful in programming languages:
  - to enforce program correctness
  - to ease program optimization
- Various type properties: type safety, weak vs strong type checking
- Typing rules can be (formally) specified using type systems
  - helps to verify the rule consistency, soundness & completeness
  - helps to implement the type-checker
- Type systems are also useful to specify/check more general program properties