



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

Types in Programming Languages

How to Formalize a Type System?

Type system for the **While** language and its extensions

Type System for other language features

Some Implementation Issues

Conclusion

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Outline: Types and Type Analysis

Types in Programming Languages

How to Formalize a Type System?

Type system for the **While** language and its extensions

Type System for other language features

Some Implementation Issues

Conclusion

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

1 | 66

Types and Type Analysis

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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, ...

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

2 | 66

Types and Type Analysis

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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 ;
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

3 | 66

Types and Type Analysis

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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).

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

4 | 66

Types and Type Analysis

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

5 | 66

Types and Type Analysis

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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>]

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

6 | 66

Types and Type Analysis

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

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
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

7 | 66

Univ. Grenoble Alpes, UFR IM²AG, 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 inferred). □

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

8 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

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...

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

9 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Outline: Types and Type Analysis

Types in Programming Languages

How to Formalize a Type System?

Type system for the While language and its extensions

Type System for other language features

Some Implementation Issues

Conclusion

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

10 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

11 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

How to Formalize a Type System?

- ▶ **Abstract syntax** describes **terms** (represented by ASTs).
- ▶ **Environment** Γ : $Name \xrightarrow{\text{part.}} \text{Type}$.
- ▶ **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
$\frac{\Gamma_1 \vdash \mathcal{A}_1 \quad \dots \quad \Gamma_n \vdash \mathcal{A}_n}{\Gamma \vdash \mathcal{A}}$	$\Gamma \vdash \mathcal{A}$

Remark A type system is an inference system. □

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

12 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Example: natural numbers

$$e ::= n \mid x \mid e_1 + e_2 \quad \text{Syntax}$$

$$\frac{\Gamma(x) = \text{Nat}}{\Gamma \vdash x : \text{Nat}} \quad x \text{ is of type Nat in environment } \Gamma \text{ if } \Gamma(x) = \text{Nat}.$$

$$\overline{\Gamma \vdash n : \text{Nat}} \quad \text{The denotation } n \text{ is of type Nat.}$$

$$\frac{\Gamma \vdash e_1 : \text{Nat} \quad \Gamma \vdash e_2 : \text{Nat}}{\Gamma \vdash e_1 + e_2 : \text{Nat}} \quad e_1 + e_2 \text{ is of type Nat assuming that } e_1 \text{ and } e_2 \text{ are of type Nat.}$$

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

13 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

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

$$\frac{\emptyset \vdash 1 : \text{Nat} \quad \emptyset \vdash 2 : \text{Nat}}{\emptyset \vdash 1 + 2 : \text{Nat}}$$

Exercise
Prove that $[x \rightarrow \text{Nat}, y \rightarrow \text{Nat}] \vdash x + 2 : \text{Nat}$.

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

14 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Outline: Types and Type Analysis

Types in Programming Languages

How to Formalize a Type System?

Type system for the **While** language and its extensions
 Type system of **While** (without blocks and procedures)
 Extension of the type system for **Proc**

Type System for other language features

Some Implementation Issues

Conclusion

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

15 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Outline: Types and Type Analysis

Types in Programming Languages

How to Formalize a Type System?

Type system for the **While** language and its extensions
 Type system of **While** (without blocks and procedures)
 Extension of the type system for **Proc**

Type System for other language features

Some Implementation Issues

Conclusion

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

16 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

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

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

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 $\{+, -, \times, \dots\}$, opb to be $\{\text{and}, \text{or}, \dots\}$, and oprel to be $\{=, <, >, \neq, \dots\}$. For example:

- 42
- $x1$
- $2 \times x2$
- $x1 = x2$
- false
- $x1 + 42$
- $xb \text{ and } x2 > 0$
- $x1 > x2$

are expressions obtained with the above grammar.

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

17 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Syntax of Language **While**

Statements

$$S ::= x := e \quad (\text{assignment of an expression to a variable } x)$$

$$\quad \text{skip} \quad (\text{doing nothing})$$

$$\quad S ; S \quad (\text{sequential composition})$$

$$\quad \text{if } e \text{ then } S \text{ else } S \text{ fi} \quad (\text{conditional composition})$$

$$\quad \text{while } e \text{ do } S \text{ od} \quad (\text{iterative and unbounded composition})$$

Statements

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

```

skip
...
x1 := 42 + y;
if (x1 + z > 0) then
...
while (x > 0) do
  y := y + z;
  x := x - 1;
od
else
  x1 := 42;
  y := 0;
fi
x1 := 42;
y := x1 + 12

```

are statements obtained with the above grammar.

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

18 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

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 ".

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

19 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Type System for Expressions

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

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

20 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Type system for Statements

Assignment	Skip
$\frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t}{\Gamma \vdash x := e}$	$\frac{}{\Gamma \vdash \text{skip}}$

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

21 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

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:

- Type conversions are never allowed.
- Only explicit conversions (with a **cast** operator) are allowed.
- (implicit) conversions are allowed.

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

22 | 66

Programs

Definition 4 (Program abstract syntax)

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

$$\begin{aligned} C &::= \text{global } D_V \text{ in } S \\ D_V &::= x : t, D_V \mid \varepsilon \end{aligned}$$

where:

- ▶ $D_V = x_1 : t_1, \dots, x_n : t_n$ ($n \geq 0$) are the **global variables**, where:
 $x_i : t_i \in \mathbf{Var} \times \mathbf{Type}$ 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

- ▶ $\mathbf{vars}(D_V)$ 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

- ▶ $\Gamma \vdash D_V \mid \Gamma_I$ means
"declarations D_V update environment Γ into Γ_I "
- ▶ $\Gamma \vdash S$ 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

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

$x \notin \mathbf{vars}(D_V)$ 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

$$\frac{\Gamma \vdash e \mid \Gamma \quad \Gamma[x \mapsto t] \vdash D_V \mid \Gamma_I}{\Gamma \vdash x : t ; D_V \mid \Gamma_I[x \mapsto t]}$$

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

Types in Programming Languages

How to Formalize a Type System?

Type system for the **While** language and its extensions
 Type system of **While** (without blocks and procedures)
 Extension of the type system for **Proc**

Type System for other language features

Some Implementation Issues

Conclusion

(Parameterless) Procedure definitions

Definition 5 (Syntactic categories of Proc)

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

Definition 6 (Syntax of procedure definitions)

The syntax of procedure definitions $D_P \in \mathbf{Pdef}$ is as follows:

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

where:

- ▶ $F \in \mathbf{Pid}$ is the **procedure identifier**.
- ▶ $S \in \mathbf{Stm}$ is the **procedure body**.
- ▶ $D_V = x_1 : t_1, \dots, x_n : t_n \in \mathbf{Var} \times (n \geq 0)$ 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 $(C, C', C_0, \dots)$ is defined as follows:

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

where:

- ▶ $D_V = x_1 : t_1, \dots, x_n : t_n$ ($n \geq 0$) are the **global variable declarations**
- ▶ $D_{Proc} = D_1, \dots, D_m \in \mathbf{Pdef}$ ($m \geq 0$) 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 $\Gamma_P : \mathbf{Pid} \rightarrow \{\text{proc}\}$
- ▶ $\mathbf{procs}(D_P)$ 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 $x \in \text{Dom}(\Gamma_2)$
 - ▶ $\Gamma'(x) = \Gamma_1(x)$ if $x \in \text{Dom}(\Gamma_1) \setminus \text{Dom}(\Gamma_2)$
- ▶ $\Gamma_V \vdash D_V \mid \Gamma'_V$ 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 ."*

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

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

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

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

Inference rule for procedure calls

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 32 | 66

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

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
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 33 | 66

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

Procedures with parameters

Definition 8 (Updated Syntax of procedure definitions)

The syntax of procedure definitions $D_P \in \mathbf{Pdef}$ is as follows:

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

where:

$D_F = y_1 : t_1, \dots, y_n : t_n \in \mathbf{Var} \times \mathbf{Type} (n \geq 0)$ is the **formal parameters declaration**.
(if $n = 0$ then the parenthesis may be omitted)

Definition 9 (Updated Syntax of procedure calls)

$$S ::= \dots \mid F(e_1, \dots, e_n)$$

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
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 34 | 66

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

Programs with procedures

Definition 10 (Program abstract syntax (unchanged))

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

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

where:

- $D_V = x_1 : t_1, \dots, x_n : t_n \ (n \geq 0)$ are the **global variable declarations**
- $D_{Proc} = D_1, \dots, D_m \in \mathbf{Pdef} \ (m \geq 0)$ 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)
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 35 | 66

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

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:

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

Proposed answers (compatible with NOSI):

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

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

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 36 | 66

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

Extending (once again !) the Type System

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

Inference rules for procedure declarations

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

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

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

Inference rule for procedure calls

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

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 37 | 66

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

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
- mutually 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 ...

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 38 | 66

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

Outline: Types and Type Analysis

- Types in Programming Languages
- How to Formalize a Type System?
- Type system for the While language and its extensions
- Type System for other language features
- Some Implementation Issues
- Conclusion

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr) 39 | 66

A Small Functional Language

Syntax of the language

$$e ::= n \mid r \mid \mathbf{true} \mid \mathbf{false} \mid x \mid \mathbf{fun} \ x : \tau. e \mid (e \ e) \mid (e \ , \ e)$$

$$\tau ::= \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{Real} \mid \tau \rightarrow \tau \mid \tau \times \tau$$

Example 14 (Programs)

- 42
- (x 12.5)
- (x , true)
- fun x : Bool. x
- ((fun x : Bool. x) 12)
- fun x : Int → Real. (x 12)

Version 1: no polymorphism, explicit type annotations

Judgment

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

Type System

$$\overline{\Gamma \vdash n : \mathbf{Int}} \quad \overline{\Gamma \vdash r : \mathbf{Real}} \quad \overline{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \quad \overline{\Gamma \vdash \mathbf{false} : \mathbf{Bool}}$$

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

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

Extension: definition of identifiers

We add a new construct:

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

Informal semantics:

within e_2 , each occurrence of x is replaced by e_1

Extending the type system to handle identifiers

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

Version 2: no polymorphism, no type annotations

Syntax of the language

$$e ::= \dots \mid \mathbf{fun} \ x. e \mid \mathbf{let} \ x = e_1 \mathbf{in} \ e_2$$

Modified type system

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

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

⇒ a unique value for type τ_1 has to be inferred ...

Examples

Expressions that can be typed:

- ((fun x.x) 1) : Int
- ((fun x.x) true) : Bool
- let x = 1 in ((fun y.y) x) : Int
- let f = fun x.x in (f 2) : Int

Expressions that cannot be typed

$\nexists (\Gamma, \tau)$ such that $\Gamma \vdash e : \tau$

- (1 2)
- fun x.(x x)
- let f = fun x.x in ((f 1) , (f true))

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 :

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

$$\begin{aligned} \mathcal{D}(\alpha) &= \{\alpha\} \\ \mathcal{D}(\tau_1 \rightarrow \tau_2) &= \mathcal{D}(\tau_1) \cup \mathcal{D}(\tau_2) \\ \mathcal{D}(\forall \alpha. \tau) &= \mathcal{D}(\tau) \setminus \{\alpha\} \\ \mathcal{D}(\Gamma) &= \bigcup_{x \in \mathbf{dom}(\Gamma)} \mathcal{D}(\Gamma(x)) \end{aligned}$$

Polymorphism: the F system

Definition 12 (Rules for system F)

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

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

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. ...



Polymorphism: the Hindley-Milner system

Type quantifiers may only appear “in front” of type expressions.

Definition 13 (New Syntax)

Types $\tau ::= \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{Real} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \alpha$
Type patterns $\sigma ::= \tau \mid \forall \alpha. \sigma$

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

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

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

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

Example 17

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

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

A "Python-like" Language

Syntax of the language

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

$$S ::= \dots \mid \text{begin } S \text{ end}$$

↔ the type of a variable is inferred 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
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

48 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

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
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

49 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

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' "

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

50 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Solution 1 (static typing): rules

Assignment	Skip
$\frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t}{\Gamma \vdash x := e \mid \Gamma[x \rightarrow t]}$	$\frac{}{\Gamma \vdash \text{skip} \mid \Gamma}$

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

51 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

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:=y*2 ;
    y:=true
  else
    y:=y+1
  fi ;
  x:=y*2
end
```

x is always false

y becomes a Bool

y remains an Int

y is no longer defined in Gamma ...

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

52 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Solution 2: dynamic typing - Judgments

Clue: performs the type checking at runtime

↔ needs to mix dynamic semantic and type checking rules !

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

Example 21

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

53 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Solution 2: dynamic typing - Rules

Skip and Assignment

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

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

Sequential Composition

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

54 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Solution 2: dynamic typing - Rules (continued)

Conditional Statements

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

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

Exercise 1

Rule for the iterative statement ?

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

55 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Beyond "pure" type checking ?

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

Example: un-initialized variables in Java

each variable should be **defined** (i.e., assigned) before being **used** ($\hookrightarrow$ **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'

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

56 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Outline: Types and Type Analysis

- Types in Programming Languages
- How to Formalize a Type System?
- Type system for the While language and its extensions
- Type System for other language features
- Some Implementation Issues
- Conclusion

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

57 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Reminder

Several issues to be handled during static semantic analysis:

- type-check the input AST
 - formal specification = a **type system**
 - notion of **environment** (name binding), to be computed:
$$\begin{aligned} \Gamma_v : Name &\rightarrow Type \\ \Gamma_p : Name &\rightarrow \{proc\} \end{aligned}$$
- decorate this AST to prepare code generation
 - give a type to intermediate nodes
 - indicate implicit **type conversions**

$\Rightarrow$ How to go from type system to algorithms?

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

58 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Example

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

Type indications provided by:

- lexical analysis
- environment
- type checking

Final AST

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

59 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

From a Type System to Algorithms?

$\Rightarrow$ 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
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

60 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Type Checking Algorithm for Arithmetic Expressions

DENOT	BINAOP	IDF
$\Gamma \vdash n : \text{Int}$	$\frac{\Gamma \vdash e_l : T_l \quad \Gamma \vdash e_r : T_r \quad T = \text{resType}(T_l, T_r)}{\Gamma \vdash e_l \text{ binaop } e_r : T}$	$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}$

```
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);
}
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

61 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

Type Checking Algorithm for Statements

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

```
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 ;
  }
}
```

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

62 | 66

Univ. Grenoble Alpes, UFR IM²AG, MoSIG 1 PLCD - Master 1 info SLPC

Types and Type Analysis

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

Frédéric Lang & Laurent Mounier (firstname.lastname@univ-grenoble-alpes.fr)

63 | 66

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., $(\Gamma_g[\Gamma_i])[\Gamma_f]$
 - ▶ a **global table**, with a **unique qualifying id** associated to each idf:
 $\{((x, \text{global}) : \text{Int}), ((x, \text{local}, \text{foo}) : \text{Real}), ((x, \text{param}, \text{foo}) : \text{Bool})\}$
 - ▶ a dynamic **stack of local tables**, one local table per environment, ordered by priority access, and updated at each procedure entry/exit
 $\{x:\text{Int}, \dots\} \longrightarrow \{x:\text{Real}, \dots\} \longrightarrow \{x:\text{Bool}, \dots\}$

Outline: Types and Type Analysis

- Types in Programming Languages
- How to Formalize a Type System?
- Type system for the While language and its extensions
- Type System for other language features
- Some Implementation Issues
- Conclusion

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