Programming Languages and Compiler Design

Programming Language Semantics
Compiler Design Techniques

Yassine Lakhnech & Laurent Mounier

{lakhnech,mounier}@imag.fr

http://www-verimag.imag.fr/˜lakhnech

http://www-verimag.imag.fr/˜mounier.

Master of Sciences in Informatics at Grenoble (MoSIG)

Grenoble Universités

(Université Joseph Fourier, Grenoble INP)

Static Semantic Analysis

Static semantic analysis

Input: : Abstract Syntax Tree (AST)

Output: : enriched AST
(with type information and/or type conversion indications)

Two main purposes:

- name identification: $\longrightarrow$ bind **use-def** occurrences
- type verification and/or type inference

Overview

1. Types in programming languages
2. How to formalize a type system ?
3. Example 1: an imperative language
4. Example 2 : a functional language
5. Some implementation issues . . .

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, and the resulting type

What are types useful for ?

- Pgm correctness

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

- Pgm readability

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

- Pgm optimization

```
var x, y, z : integer ; -- and not real
x := y + z ; -- integer operations are used
```

Typed and untyped languages

- Typed languages:
  $\longrightarrow$ a dedicated type is associated to each identifier
  (and hence to each expression)

  examples: Java, Ada, C, Pascal, CAML, etc.
  Rk : strongly typed vs weakly typed languages . . .

- Untyped languages:
  $\longrightarrow$ A single (universal) type is associated to each identifier
  (and hence to each expression)

  examples: Assembly language, shell-script, Lisp, etc.

Typed languages vs and safe languages

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

(Robin Milner)

Trapped errors vs untrapped errors

Safe language = untrapped errors are not possible

Using types in programming languages is a way to ensure safety but:

- it is not the only one (Lisp is considered safe)
- it is not sufficient (C is considered unsafe)

Types and type constructions

- Basic types: integers, boolean, characters, etc.
- Type constructions
  - cartesian product (structure)
  - 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.

Examples:

- 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
```

Type checking vs type inference

In a typed language, the set of “correct typing rules” is called a 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 pgm

Type inference: compute a consistent type for each pgm fragments

Rk: in some languages (e.g., CAML, Haskel), there are no type annotations at all (all types are infered).

Static checking vs dynamic checking

- static checking: verification performed at compile-time
- dynamic checking: verification performed at run-time
  $\longrightarrow$ necessary to correctly handle:
  - dynamic binding for variables or procedures
  - polymorphism
  - array bounds
  - subtyping
  - etc.

$\Rightarrow$ For most programming languages, both kinds of checks are used . . .

How to formalize a type system ? (1)

- “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 on its **free variables** ...

How to formalize a type system ? (2)

- Abstract syntax describes terms (representing AST)
- Environment Γ: $Name \rightarrow Types$ (partial)
- Judgement $\Gamma \vdash t : \tau$

  In the environnemt Γ, the term t is well-typed and has type τ.

  (free variables of t belong to the domain of Γ)
- Type system

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

Example: natural numbers

$$a \quad := \quad n \mid x \mid a_1 + a_2$$

Syntax

$$\frac{}{\Gamma \vdash x : \textbf{Nat}} \ (\text{if } \Gamma(x) = \textbf{Nat})$$

x is of type **Nat** in the environment Γ if $\Gamma(x) = \textbf{Nat}$.

$$\frac{}{\Gamma \vdash n : \textbf{Nat}}$$

The denotation n is of type **Nat**

$$\frac{\Gamma \vdash a_1 : \textbf{Nat} \quad \Gamma \vdash a_2 : \textbf{Nat}}{\Gamma \vdash a_1 + a_2 : \textbf{Nat}}$$

$a_1 + a_2$ is of type **Nat** assuming that a_1 and a_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 jugement is valid iff it is the root of a derivation tree
example:

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

exo: prove that $[x \rightarrow Nat, y \rightarrow Nat] \vdash x + 2 : Nat$

Type system for the `while` *language*

Syntax of the `while` *language*

Expressions

- same syntax for boolean and integer expressions (a)
- 3 kinds of (syntacticaly) distinct binary operators: arithmetic (opa), boolean (opb) and relational (`oprel`)

$$\mathtt{a} \quad ::= \quad \mathtt{true} \mid \mathtt{false} \mid \mathtt{n} \mid \mathtt{x} \mid \mathtt{a\ opa\ a} \mid \mathtt{a\ oprel\ a} \mid \mathtt{a\ opb\ a}$$

Statements

$$\begin{array}{lll} \mathsf{S} & ::= & \mathsf{x := a} \mid \mathsf{skip} \mid \mathsf{S\ ;\ S} \mid \\ & & \mathtt{if\ a\ then\ S\ else\ S} \mid \mathtt{while\ \ a\ do\ S} \end{array}$$

Judgments

- $\Gamma \vdash S$
  "in the environment Γ the statement S is well-typed".

- $\Gamma \vdash a : t$
  "in the environment Γ the expression a is of type t.

Type system for expressions

bool. constant	**int. constant**	**int opbin**
$\dfrac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}}$	$\dfrac{}{\Gamma \vdash \texttt{n} : \texttt{Int}}$	$\dfrac{\Gamma \vdash a_1 : \texttt{Int} \quad \Gamma \vdash a_2 : \texttt{Int}}{\Gamma \vdash a_1 \ \texttt{opa} \ a_2 : \textbf{Int}}$

variables	**bool. opbin**	**relational operators**
$\dfrac{\Gamma(x)=t}{\Gamma \vdash x : t}$	$\dfrac{\Gamma \vdash b_1 : \textbf{Bool} \quad \Gamma \vdash b_1 : \textbf{Bool}}{\Gamma \vdash b_1 \ \texttt{opb} \ b_2 : \textbf{Bool}}$	$\dfrac{\Gamma \vdash a_1 : t \quad \Gamma \vdash a_2 : t}{\Gamma \vdash a_1 \ \texttt{oprel} \ a_2 : \textbf{Bool}}$

Type system for statements

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

Sequence	**Iteration**
$\dfrac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1 ; S_2}$	$\dfrac{\Gamma \vdash a : \mathbf{Bool} \quad \Gamma \vdash S}{\Gamma \vdash \texttt{while} \ \ a\ \texttt{do}\ S}$

Exercise: conditional statement ?

Exercise

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

Extension 1: Blocks

A new syntactic rule for the statements:

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

And for the declarations:

$$D_V ::= \textbf{var } x := a \; ; \; D_V \mid \epsilon$$

Notations

- $\mathtt{DV}(D_v)$ 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 environnement Γ into Γ_l
- $\Gamma \vdash \; S$ means

  statement S is well-typed within environnement Γ

Inference rule for Blocks

$$\frac{\Gamma \vdash \ D_V \mid \Gamma_l \qquad \Gamma_l \vdash S}{\Gamma \vdash \textbf{begin}\ D_V\ ;\ S\ \textbf{end}}$$

Inference rules for declarations

Sequential evaluation

$$\frac{}{\Gamma \vdash \epsilon \mid \Gamma} \qquad \frac{\Gamma \vdash a : t \quad \Gamma[x \mapsto t] \vdash D_V \mid \Gamma_l \quad x \notin \mathsf{DV}(D_V)}{\Gamma \vdash \textbf{var } x := a \; ; \; D_V \mid \Gamma_l[x \mapsto t]}$$

Collateral evaluation

$$\frac{}{\Gamma \vdash \epsilon \mid \Gamma} \qquad \frac{\Gamma \vdash a : t \quad \Gamma \vdash D_V \mid \Gamma_l \quad x \notin \mathsf{DV}(D_V)}{\Gamma \vdash \textbf{var } x := a \; ; \; D_V \mid \Gamma_l[x \mapsto t]}$$

Possible variations for variable declarations

- explicitely typed variables:
  `var x := e :  t`
- uninitialized variables:
  `var x :t`
- untyped variables ?
  `var x := e`
- uninitialized and untyped variables ???
  `var x`

Extension 2: Procedures

Syntactic rule for the statements:

$$S ::= \cdots \mid \textbf{begin}\ D_V\ ; D_P\ ;\ S\ \textbf{end} \mid \textbf{call}\ p$$

and for the declarations:

$$D_P ::= \textbf{proc}\ p\ \textbf{is}\ S\ ;\ D_P \mid \epsilon$$

$DP(D_P)$ denotes the set of procedures **declared** in D_P

Reminder: the semantics depends on the kind of binding (static vs dynamic) you consider . . .

Judgements

- procedure environment $\Gamma_P : Name \to \{proc\}$ (partial)
- $\Gamma_V \vdash\ D_V \mid \Gamma'_V$ means

  variable declarations D_V update variable environment Γ_V into Γ'_V
- $(\Gamma_V, \Gamma_P) \vdash\ D_P$ means

  procedure declarations D_P is well-typed within variable and procedure environments (Γ_V, Γ_P)
- $(\Gamma_V, \Gamma_P) \vdash\ S$ means

  statement S is well-typed within variable and procedure environments (Γ_V, Γ_P)

Static binding for proc. and var.

Block $$\frac{\Gamma_V \vdash D_V \mid \Gamma'_V \quad (\Gamma'_V, \Gamma_P) \vdash D_P \quad (\Gamma'_V, \Gamma'_P) \vdash S}{(\Gamma_V, \Gamma_P) \vdash \textbf{begin } D_V \ ; \ D_P \ ; \ S \textbf{ end}}$$

D_P $$\frac{(\Gamma_V, \Gamma_P) \vdash \ S \quad (\Gamma_V, \Gamma_P[p \mapsto \textbf{proc}]) \vdash \ D_P \quad p \notin DP(D_P)}{(\Gamma_V, \Gamma_P) \vdash \textbf{proc } p \textbf{ is } S \ ; \ D_P}$$

Call $$\frac{}{(\Gamma_V, \Gamma_P) \vdash \ \textbf{call } p} \ \Gamma_P(p) = \texttt{proc}$$

where $\Gamma'_P = \mathsf{upd}(\Gamma_P, D_P)$
with :

$$\begin{aligned} \mathsf{upd}(\Gamma_P, \textbf{proc } p \textbf{ is } S \ ; \ D_P) &= \mathsf{upd}(\Gamma_P[p \mapsto \textbf{proc}], D_P) \\ \mathsf{upd}(\Gamma_P, \varepsilon) &= \Gamma_P \end{aligned}$$

Dynamic binding for proc. and var.

Block $$\frac{\Gamma_V \vdash D_V \mid \Gamma'_V \quad (\Gamma'_V, \Gamma'_P) \vdash S \quad \texttt{udef}(D_P)}{(\Gamma_V, \Gamma_P) \vdash \textbf{begin } D_V \; ; \; D_P \; ; \; S \textbf{ end}}$$

Call $$\frac{(\Gamma_V, \Gamma_P) \vdash S}{(\Gamma_V, \Gamma_P) \vdash \textbf{ call } p} \Gamma_P(p) = S$$

where $\Gamma'_P = \texttt{upd}(\Gamma_P, D_P)$
with:

$$\begin{array}{rcl}
\texttt{upd}(\Gamma_P, \textbf{proc } p \textbf{ is } S \; ; \; D_P) & = & \texttt{upd}(\Gamma_P[p \mapsto S], D_P) \\
\texttt{upd}(\Gamma_P, \varepsilon) & = & \Gamma_P \\
\texttt{udef}(\textbf{proc } p \textbf{ is } S \; ; \; D_P)) & = & \texttt{udef}(D_P) \wedge p \notin DP(D_P) \\
\texttt{udef}(\varepsilon) & = & \text{true}
\end{array}$$

Remark :

procedure environment $\Gamma_P : Name \rightarrow Stm$ (partial)

Procedures: static ; variables: dynamic

Block $$\frac{(\Gamma_V, D_V) \longrightarrow \Gamma'_V \quad (\Gamma'_V, \Gamma_P) \vdash D_P \quad (\Gamma'_V, \Gamma'_P) \vdash S}{(\Gamma_V, \Gamma_P) \vdash \textbf{begin } D_V \; ; \; D_P \; ; \; S \textbf{ end}}$$

Call $$\frac{(\Gamma_V, \Gamma'_P) \vdash S}{(\Gamma_V, \Gamma_P) \vdash \textbf{ call } p} \Gamma_P(p) = (\Gamma'_P, S)$$

where $\Gamma'_P = \mathsf{upd}(\Gamma_P, D_P)$
with:

$$\begin{aligned} \mathsf{upd}(\Gamma_P, \textbf{proc } p \textbf{ is } S \; ; \; D_P) &= \mathsf{upd}(\Gamma_P[p \mapsto (\Gamma_P, S)], D_P) \\ \mathsf{upd}(\Gamma_P, \varepsilon) &= \Gamma_P \end{aligned}$$

Remark :

$ProcEnv : Name \rightarrow ProcEnv \times Stm$ (partial)

$\Gamma_P \in ProcEnv$

Exercices

What about recursive procedures ?

Type system for a (small) functional language

A small functional language

Syntax

$$e \quad ::= \quad n \mid r \mid \textbf{true} \mid x \mid \textbf{fun}\ x : \tau.e \mid (e\ e) \mid (e\ ,\ e)$$
$$\tau \quad ::= \quad \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} \mid \tau \rightarrow \tau \mid \tau \times \tau$$

Examples

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

Version 1: no polymorhism, explicit type annotations

Judgement

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

Type System

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

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

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

Extension

We add a new construct:

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

Informal semantics:

within e_2, each occurence of x is replaced by e_1

Type System

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

Version 2: no polymorphism, no type annotations

New Syntax

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

Modified rules

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

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

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

Examples

Expressions that can be typed:

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

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

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

Polymorphism ?

We introduce:

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

example: **fun** $x.x$ is of type $\forall\alpha.\alpha \rightarrow \alpha$

Set of free type variables of an environment Γ:

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

$$\mathcal{D}(\alpha) = \{\alpha\}$$

$$\mathcal{D}(\tau_1 \longrightarrow \tau_2) = \mathcal{D}(\tau_1) \cup \mathcal{D}(\tau_2)$$

$$\mathcal{D}(\forall\alpha \cdot \tau) = \mathcal{D}(\tau) \setminus \{\alpha\}$$

$$\mathcal{D}(\Gamma) = \bigcup_{x \in \textbf{dom}(\Gamma)} \mathcal{D}(\Gamma(x))$$

Polymorphism: the F system

2 new rules

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

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

examples:

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

Rk: type inference is no longer decidable in this type system . . .

Polymorphism: Hindley-Milner system

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

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

3 rules are modified:

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

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

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

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

Some implementation issues

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:

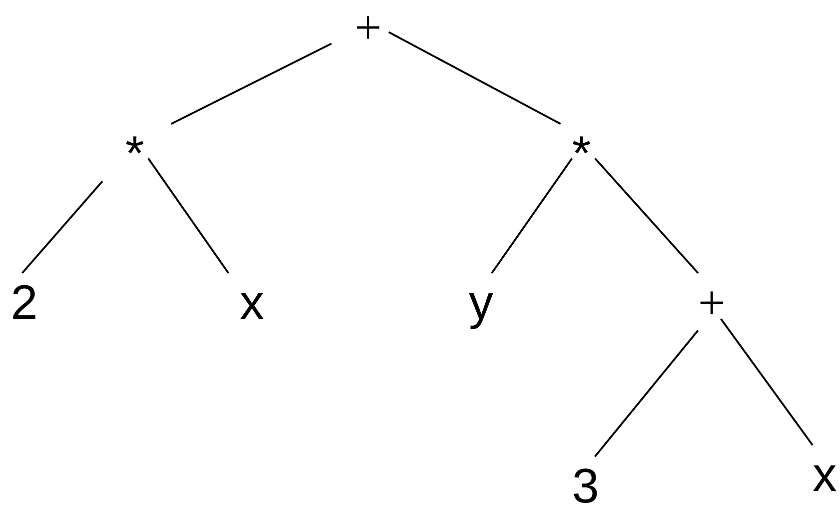
$$\Gamma_V : Name \rightarrow Type$$
$$\Gamma_P : Name \rightarrow \{proc\}$$

2. decorate this AST to prepare code generation
   - give a type to intermediate nodes
   - indicate implicit type conversions

$\Rightarrow$ from type system to algorithms ?

Example (1)

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



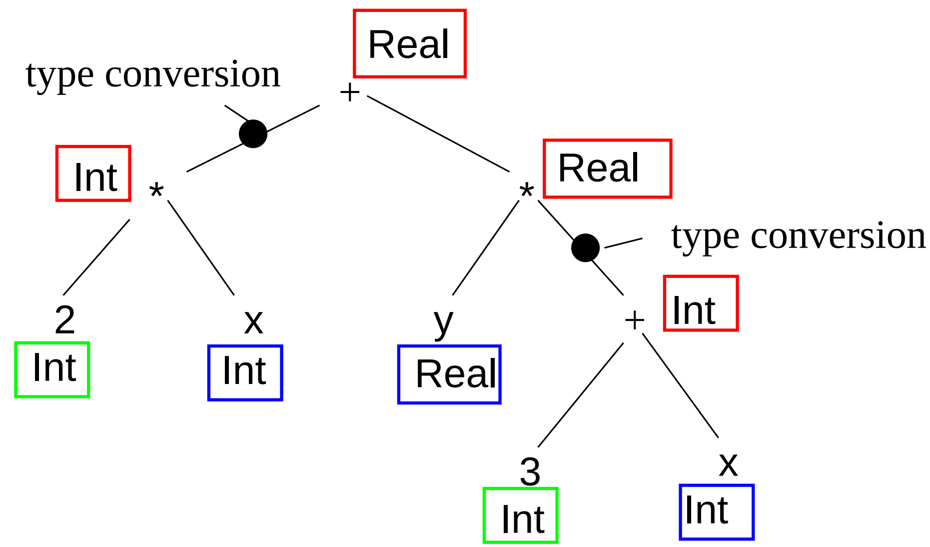
Initial AST

Example (2)

```
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

From 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
```

Type checking algorithm for arithmetic expressions

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

```
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
$\dfrac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1 ; S_2}$	$\dfrac{\Gamma \vdash b : \mathbf{Bool} \quad \Gamma \vdash S}{\Gamma \vdash \texttt{while} \;\; b \; \texttt{do} \; S}$	$\dfrac{\Gamma \vdash x : t \quad \Gamma \vdash e : t}{\Gamma \vdash x := e}$

```
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)

- handle nested declarations:

```
begin
   var x : Int ; var y : Real ;
   begin
     var x : Boolean ;
     x = y > 2.5 ;
   end
end
```

Usual solution: symbol table

- 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.

- two solutions for handling nested blocks ($\Gamma[x \rightarrow \texttt{Bool}]$)

  - a global table, with a unique id is associated to each idf:
    $\{((\texttt{x}, 1) : \texttt{Int}), ((\texttt{y}, 1) : \texttt{Real}), ((\texttt{x}, 1.1) : \texttt{Bool})\}$
    $\rightarrow$ based on a unique (hierarchical) numbering of blocks

  - a dynamic stack of local tables, one local table per block:
    $\{\texttt{x:Int, y:Real}\} \longrightarrow \{\texttt{x:Bool}\}$