Static (Semantic) Analysis

(Third) Last Step of Compiler Front-End

Compositional and Contextual Property Analysis

Compositional and Contextual Property Analysis

- **Main Properties**:
  - Uniqueness
  - Well formed (control) Structures
  - Correlated Occurrences
- **Types:**
  - Type Systems
  - Type Checking
  - Type Inference

Uniqueness - Control Structure

Uniqueness

- no name collision (for instance, in a block, or definition..)

Well Formed Structures

- Checkings for correct use of construct compositions:
  - in C, *break* may only, occur inside blocks;
  - in Java, no *hiding variable* is permitted in blocks
  - in Java, classes implementing interface must contain definitions for the interface methods
  - in Pascal, the *for-block* cannot modify *for-index*
  - Expressions used as *by-reference* parameters must have *l-values*

  ….

Correlated Occurrences

Correlated Occurrences

- Specific Checkings for the correct use of the constructs:
  - A *declared identifier* must occur in some use
  - In many languages, a used identifier must be declared with the right scope.
  - in Pascal, the *function body* must contain an assignment to the function name;
  - In C, non-void procedure bodies must contains *return exp*

Types

Are a Special Kind of Terms that:

- *are assigned* to the program structures
- *are fundamental for classifying* program structures with the aim of:
  - studying (semantic) correctness of the structure use
  - preventing run-time errors
  - allow code optimizations at compile/run-time
- *are expressed* by a suitable set of expressions:
  - called **Type Expressions** and
  - are obeying the laws of a specific system, called **Type System**

Type Expressions

1) **Basic (or Atomic) Types**

real, int, char, file, unit

2) *Type Constructors for* **Derived Types**

array: I x T -> array(I,T)
product: T1 x T2 -> T1 ✖ T2
record: ({i1}xT1)x..x({ik}xTk) -> record(i1:T1...ik:Tk)
enumerated: {v1,..,vk} -> (v1,..,vk)
pointer: T -> pointer(T)
function: TD x TC -> TD ➔ TC
procedure: TD -> TD ➔ unit

3) **Type Identifier** (for naming)

4) **Type Variables** (for polimorphic types)

Type Checking (2)

- **Associating Types to Program Structures: Typing Rules**
- **Extending the attribute plans of the previous Grammars**
- Derived Types: Typing Rules
- Coercion and overloading: Typing Rules
- Extending the attribute plans of the previous grammars

Assigning Types to Program Structures of a Strongly Typed Language

Types

Types=Basic-Types + N +Types X Types ->Types

Rules

$$\frac{\Gamma\text{|- i:t}\quad \Gamma\text{|- e:T}}{\Gamma\text{|- i:=e:N}} \qquad \frac{\Gamma\text{|-e:boolean}\quad \Gamma\text{|-Cs: N}}{\Gamma\text{|- while e do Cs :N}} \qquad \frac{\Gamma\text{|-C:N}\quad \Gamma\text{|-Cs:N}}{\Gamma\text{|-C;Cs : N}}$$

$$\frac{}{\Gamma_1,i:T,\Gamma_2\text{|- i:T}} \qquad \frac{\Gamma\text{|-e1:T1}\quad \Gamma\text{|-e2: T2}\quad \Gamma\text{|-op:T1xT2->T}}{\Gamma\text{|- op(e1,e2):T}} \qquad \frac{\Gamma,i:Ti,j:Tj\text{ |-e:T}}{\Gamma\text{|-fun(i:Ti,j:Tj)\{e\} : TixTj->T}}$$

Attributes:

r:- type of all the program structures but expressions
-synthesized of **P,C,Cs,A,W**

type:- type of expression
-synthesized of **E,F, E',F', T, ide**

in: -type expression of the left brother on the left
-inherited of **E', F'**

It is extending the first grammar in its last variant discussed in variant

A type checker for Simple Commands

[0] P::= Ds Cs	P.r:= Cs.r
[1] P::= Cs	P.r:= Cs.r
[2] Ds::= Var Dts	
[3] Dts::= Dt Dts'	
[4] Dts'::= ; Dt Dts'	
[5] Dts'::= ε	
[6] Dt::= ide O	addtype(ide.entry,O.t)
[7] O_1::= , ide O_2	addtype(ide.entry,O_2.t); O_1.t:= O_2.t
[8] O::= : Y	O.t:=Y.t
[9] Cs_1::= ; C Cs_2	Cs_1.r:= if (C.r=N & Cs_1.r=N) then N else ⊥
[10]Cs::= ε	Cs.r::= N
[11]C::= A	C.r::= A.r
[12]C::= W	C.r::= W.r
[13]A::= ide:=E	A.r::= if (ide.type=E.type & ide.type≠⊥) then N else ⊥
[14]W::= while E do C Cs endw	W.r::=if(E.type=boolean & C.r=Cs.r) then C.r else ⊥
[24]Y::= boolean	Y.t:= boolean
[25]Y::= integer	Y.t:= integer
[26]Y::= file	Y.t:= file

A type checker for Simple Expressions

[15]E::= F E'	{E'.in = F.type; E.type := E'.type;

[16]E'_1::= op-l F E'_2	{match t1xt2->t = op-l.type;
	if (E'_1.in = t1 & F.type = t2)
	then {E'_2.in = t; E'_1.type := E'_2.type;}
	else {E'_2.in = ⊥; E'_1.type :=⊥;};
	}

[17]E'::= ε	{E'.type = E'.in}
[18]F::= T F'	{…}
[19]F'::= op-h T F'	{…}
[20]F'::= ε	{…}
[21]T::= num	{T.type = integer;}
[22]T::= ide	{T.type = ide.type;}
[23]T::= (E)	{T.type = E.type;}

What about when Simple contains 2-arity Functional Abstraction

[0] P::= Ds Cs	{P.r:= Ds.r & Cs.r}
[1] P::= Cs	{P.r:= Cs.r}
[02]Ds::= Dv Df	{Ds.r = Df.r}
[03]Df_1::= FF Df_2	{Df_1.r:= FF.r & Df_2.r}
[04]FF::= fun ide_1(ide_2:Y_1, ide_3:Y_2) E:Y_3	{addtype(ide_2,Y_1.t); addtype(ide_3,Y_2.t);} {check:= (E.type = Y_3.t); FF.r = check; if(check) then addtype(ide_1,Y_1 x Y_2 -> Y_3);}
…	
[23]T::= (E)	{T.type = E.type;}
[24]T::= ide(E_1,E_2)	{match t1xt2->t = ide.type; if (E_1.type=t1 & E_2.type=t2) then T.type:=t else T.type:=⊥; }

What about Structuted Data Array, Records

- Also in this case, the affected structures are those of espressions.
- We have to consider the new , involved types and the rules that define the right computational behaviour of such structures

Types

Types = Basic-Types + N +Types X Types ->Types + array(Type,Type) + record({ide:Type}$^{+}$)

Rules

$$\frac{\Gamma \vdash e:array(Ti,Tv) \quad \Gamma \vdash u:Ti' \quad \Gamma \vdash Ti \approx Ti'}{\Gamma \vdash e[u]:Tv}$$

$$\frac{\Gamma \vdash e:record(i1:T1,\ldots,ij:Tj,\ldots,in:Tn)}{\Gamma \vdash e.ij:Tj}$$

What about Type Equivalence Nominal vs. Structural

Other rules that define different forms of type equivalence and other relations (included Subtyping)

Types

Types = Basic-Types + N +Types X Types ->Types +

array(Type,Type) + record({ide:Type}$^{+}$) +

Rules

$$\frac{}{\Delta \vdash \text{boolean WFT}} \qquad \ldots \qquad \frac{\Delta \vdash T,T' \text{ WFT}}{\Delta \vdash T \to T' \text{ WFT}} \qquad \ldots$$

$$\frac{\Delta \vdash T_1,T_1',\ldots,T_n,T_n' \text{ WFT} \quad \Delta \vdash y(T_1,\ldots,T_n) \text{ WFT} \quad \Delta \vdash T_i \approx T'_i \ (1 \le i \le n)}{\Delta \vdash y(T_1,\ldots,T_n) \approx y(T'_1,\ldots,T'_n)} \qquad \text{(structural)}$$

$$\frac{\Delta \vdash T,T' \text{ WFT} \quad \Delta \vdash T = T' \text{ WFT}}{\Delta \vdash T \approx T'} \qquad \text{(nominal)}$$

What about Type Coercion, Overloading, …

Type Coercion and Type Overloading are widely used forms of (ad hoc) polimorphism

Types

Types = Basic-Types + N +Types X Types ->Types +

array(Type,Type) + record({ide:Type}$^+$) +

{Type}$^+$

Rules

$$\frac{\Gamma \vdash e:T \quad \Gamma \vdash into:T\to T'}{\Gamma \vdash into(e):T'} \quad \text{(coercion)}$$

$$\frac{\Gamma \vdash e:\{T_1\to T_1',\ldots,T_n\to T_n'\} \quad \Gamma \vdash a:T_i \quad (1\le i\le n)}{\Gamma \vdash e(a):T_i'} \qquad \frac{\Delta \vdash T_1,T_1',\ldots,T_n,T_n' \text{ WFT} \quad \Delta \vdash T_i \approx T_j \quad (1\le I<j\le n)}{\Delta \vdash e:\{T_1\to T_1',\ldots,T_n\to T_n'\} \text{ WFT}}$$

(overloading)

[0] **P ::= Ds Dff Cs**	**P.r := Cs .r**
[1] **P ::= Cs**	**P.r := Cs .r**
[2] **Ds ::= Var Dts**	
[3] **Dts ::= Dt : Y Dts'**	**addtype-set(Dt.entry, Y.t)**
[4] **Dts'$_1$ ::= ; Dt : Y Dts$_2$'**	**addtype-set(Dt.entry, Y.t)**
[5] **Dts' ::= ε**	
[6] **Dt ::= ide O**	**Dt.entry := cons(ide.entry, O.entry)**
[7] **O$_1$ ::= , ide O$_2$**	**O$_1$.entry := cons(ide.entry, O$_2$.entry)**
[8] **O ::= ε**	**O.entry := emptylist**
[28] **Dff ::= Fun Df Dff'**	
[29] **Dff'$_1$ ::= ; Df Dff'$_2$**	
[30] **Dff' ::= ε**	
[31] **Df ::= funct ide FPP : Y ; P**	**overload(ide.entry, FPP.t → Y.t)**
[32] **FPP ::= (FP : Y FPP'**	**FPP.t := Y.t × FPP'.t**
[33] **FPP ::= ε**	**FPP.t := N**
[34] **FPP'$_1$ ::= , FP : Y FPP'$_2$**	**FPP'$_1$.t := Y.t × FPP'$_2$.t**
[35] **FPP' ::=)**	**FPP'.t := N**
[36] **T ::= apply ide AP**	**let S := ide.type** **in T.type := find(AP.type, S)**