ⓒ Type systems and Semantics — (partly from Ch6 - sebesta book)

— type systems

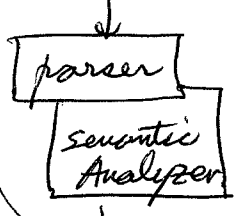
type — well defined set of values and operations on them

e.g.) int — values: {-1, -2, ..., 1, 2, 3, ...}
op: {+, -, *, /, ...}

boolean — values: {True, False}
op: {&&, ||, !}

- statically typed lang. — C, Java, ..
  - single type for a var.
  - type is determined at compile time.
  - during run time, type is not changed.
- dynamically typed lang.
  - LISP, scheme, ..
  - dynamic type binding at run time.

Static semantics
- adds context-sensitive information to the parser.
  - BNF/EBNF can't provide.



type error — operation is attempted on a value that is not well defined.

(not the matter of statically/dynamically typed)

- Strongly typed lang. — Java, ..
  - type system allows all type errors in programs to be detected either at compile time or at run time.
  - ⇒ more reliable programs.
  - type-safe programs

↓

↓

eg) C is not strongly typed lang.

```
int X;
X+1;
```
→ no compile error, no run-time error.
- eg) `int X; X=5.0;` ✗
- eg) `char X; X+1;` ✗

↑ value not assigned yet.

— dynamically typed language (e.g., LISP) must be type safe.

- type checking at run-time.
- ∴ var's type can change dynamically.

type errors

- non-unique name : eg) `int X, Y, (X);` — check in the declaration part.
- declare first, then reference : — check in statements.
- type mismatch — check in expression using type map.

BNF/EBNF notation can not express.

Issue:

How to define a type system for a language so that type errors can be detected.

— Formalizing the type system

— define a lang's type system

(write a set of func specifications (boolean func) for type safe programming. (rules)

- uniqueness of var names
- var's must be declared before referenced in statements.
- var's used in arith. expression must have proper type in declaration.

using typemap for the program:
- (checking in declaration)
- (checking in statements)
- (checking in expression)

— in the declaration part, create (build) a typemap.

— based on the typemap, static typechecking is possible.

ex) tm = { <i, int>, <j, int>, <p, boolean> }

(var. (d3.v), type (d3.t))

— func. specification for typemap building.

typing : Declarations → typeMap
(func. name; input; output)

typing : (Declarations d) = $\bigcup_{i \in \{1,\ldots,n\}} \langle d_i.v, d_i.t \rangle$ (var name, type)

⇒ implementation (in java)

```
TypeMap typing (Declaration d)
{ TypeMap map = new TypeMap();
  for (int i=0; i<d.size(); i++)
  { map.put ( ((Declaration1)(d.elementAt(i))).v,
              ((Declaration1)(d.elementAt(i))).t );
  }
  return map;
}
```

class Declaration { Variable v; Type t; }

— Static typechecking (at compile time) — func. specification

eg. V : Declarations → B (T/F)
(input: Declarations; output: boolean)

V (Declarations d) = $\forall_{i,j} \in \{1,\ldots,n\}: (i \neq j \supset d_i.v \neq d_j.v)$
(V: func. name)

if this is T → then, this is T

typechecking func. for Unique Names.

defines: every distinct pair of var's in a declaration list has mutually different identifiers.

implementation:

```
boolean V (Declarations d)
{ for (int i=0; i<d.size()-1; i++)
    for(int j=i+1; j<d.size(); j++)
      if (di.v == dj.v)
        return False;
  return True;
}
```

ref
- type of each var is in the set of available types

can be expressed in BNF/EBNF.
- Declare → Type Idlist
- Type → int | real | boolean ...

— program (P)

Declaration part (d)
body

check whole prog's type errors

typechecking func (V)

V : program → B (T/F)

V (program P) = V (P.declarepart) ∧ V (P.body, typing (P.declarepart))
(check unique ids)

V (P.body, ...): checking using typeMap

typing (P.declarepart): returns typeMap

implementation

```
boolean V (program P)
{ return V(P.declarepart) &&
         V(P.body, typing(P.declarepart));
}
```

Review

- $\underline{typing}$: (Declarations d) $= \bigcup_{i \in \{1,\ldots,n\}} \langle di.V, di.t \rangle$ (id, type)

  ($\underline{typing}$: build typeMap)

- $\underline{V}$ (Declarations d) $= \forall i,j \in \{1,\ldots,n\} : (i \neq j \supset di.v \neq dj.v)$ ($T \longrightarrow T$)

  ($\underline{V}$: check uniq. name)

- $\underline{V}$ (program p) $= \underline{V(p.declarationPart)} \wedge V(p.body, \underline{typing}(p.declarPart))$

  ($\underline{V}$: check whole prog's type errors)

  (V(p.declarationPart): check uniq. names)

  (typing(p.declarPart): returns typeMap)

  (V(p.body, ...): check type errors in the body)