Interpretação e Compilação de Linguagens (de Programação)

22/23

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Type Systems

A type system is a kind of "program logic" that ensures "good behaviour" of programs.

A type systems is a form of so-called "static analysis" or "abstract interpretation" of programs. For any syntactically correct program a static analysis always terminates, even for programs that do not terminate at runtime.

A very common static analysys technique is type checking.

A type system statically ensures (e.g. at compile time) the absence of certain kinds of runtime errors and also provides useful information for compilation.

- Runtime Errors
- Abstract Interpretation
- Type Systems and Type Checking
- Soundness of a Type Systems
- Trapping of Runtime Errors

Runtime errors …

- What may go wrong during a program execution ?

```
def
   a = new 0
   b = new 2
   c = new (!a > !b)
in
   if !c then
      a := a + 1
   else c := !b < !c
end
```

Runtime errors …

- What may go wrong during a program execution ?

```
def
  a = new 0
  b = new 2
  c = new (!a > !b)
in
  if c then
     a := !a + 1;
     c := !b < !a
end
```

Runtime errors …

- What may go wrong during a program execution ?

eval(**add(**E1, E2**)** , *env* , *m0*) ≜**[** (*v1* , *m1*) = **eval**(E1, *env, m0*);
(*v2* , *m2*) = **eval**(E2, *env, m1*);
(*v1* + *v2* , *m2*) **]**

eval(**deref**(E) , env , m0) ≜ **[** (ref , m1) = **eval**(E, env, m0);
(m1.**get**(ref) , m1) **]**

eval(**assign**(E1, E2) , env , m0) ≜ **[**(v1 , m1) = **eval**(E1, env, m0);
(v2 , m2) = **eval**(E2, env, m1);
m3 = m2.**set**(v1, v2) ;
(v1 , m3) **]**

It is not feasible during compilation to compute concrete values, and check if all operations will not go wrong…
The set of possible values is infinite

However, we may approximate such sets of values by classifying them into "types".

Semantic maps (we have seen before)

- Map **eval** to compute a value + effect for any CALCS programs:

$$\text{eval} : \text{CALCS} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Map **comp** that translates CALCS programs in JVM bytecode

$$\text{comp} : \text{CALCS} \times \text{ENV} \rightarrow \text{CodeSeq}$$

Typechecker (as an abstract interpreter)

- Map **eval** to compute a value + effect for any CALCS programs:

$$\mathrm{eval} : \mathrm{CALCS} \times \mathrm{ENV} \times \mathrm{MEM} \rightarrow \mathrm{VAL} \times \mathrm{MEM}$$

- Map **comp** that translates CALCS programs in JVM bytecode

$$\mathrm{comp} : \mathrm{CALCS} \times \mathrm{ENV} \rightarrow \mathrm{CodeSeq}$$

- Map **typchk** that computes a type for a CALCS program:

$$\mathrm{typechk} : \mathrm{CALCS} \times \mathrm{ENV} \rightarrow \mathrm{TYPE} \cup \{ \mathrm{TYPEERROR} \}$$

$$\mathrm{ENV} : \mathrm{ID} \rightarrow \mathrm{TYPE}$$

$$\mathrm{TYPE} = \{ \mathrm{int}, \mathrm{bool}, \mathrm{ref}\{\mathrm{TYPE}\} \}$$

Typechecker (as an abstract interpreter)

- Map **eval** to compute a value + effect for any CALCS programs:

$$\text{eval} : \text{CALCS} \times \text{ENV}\langle\text{IValue}\rangle \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

- Map **comp** that translates CALCS programs in JVM bytecode

$$\text{comp} : \text{CALCS} \times \text{ENV}\langle\text{Coord}\rangle \rightarrow \text{CodeSeq}$$

- Map **typchk** that computes a type for any CALCS program:

$$\text{typechk} : \text{CALCS} \times \text{ENV}\langle\text{TYPE}\rangle \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

$$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

$$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\}, \text{none} \}$$

Types for CALCS

- Map **typchk** that computes a type for any CALCS program:

  $$\text{typechk} : \text{CALCS} \times \text{ENV}\langle\text{TYPE}\rangle \rightarrow \text{TYPE} \cup \{ \text{TYPEERROR} \}$$

  $$\text{ENV} : \text{ID} \rightarrow \text{TYPE}$$

  $$\text{TYPE} = \{ \text{int}, \text{bool}, \text{ref}\{\text{TYPE}\} \}$$

- Map **typchk** may be understood as an “abstract” interpreter that evaluates the program according to a special semantics, more abstract, in which values are approximated by types.

- In the abstract sematics, the values are the types and operations compute types from types.

Types for CALCS

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

  TYPE = { int, bool, ref{TYPE} }

int: type of integer values.

bool: type of boolean values.

ref{$\mathcal{T}$}: type of references that may only hold values of type $\mathcal{T}$.

Example: ref{ref{int}} is the type of references that may only hold references that may only hold integers.

Types for CALCS (defined by a grammar)

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

  TYPE = { int, bool, ref{TYPE} }

int: type of integer values.

bool: type of boolean values.

ref{$\mathcal{T}$}: type of references that may only hold values of type $\mathcal{T}$.

Type → int | bool | ref [Type]

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( add(E1, E2) , env ) ≜ [ t1 =  typchk ( E1, env)
                                t2 =  typchk ( E2, env)
                                if  (t1 == int ) and (t2 == int )
                                     then int
                                     else TYPEERROR ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

```
typchk( num(n) , env)    ≜ int ;
typchk( id(s) , env)     ≜ env.Find(s) ;
typchk( true , env)      ≜ bool ;
typchk( false , env)     ≜ bool ;
```

  – All integers num(n) are "evaluated " into int.
  – All booleans are "evaluated " into bool.

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

```
typchk( def(s, E1, E2), env) ≜ [ ts = typchk(E1, env);
                                 env = env.BeginScope();
                                 env =  env.Assoc(s, ts);
                                 t = typchk(E2, env);
                                 env = env.EndScope();
                                 return t ]
```

  - Associate each names s to a type ts of initialiser **E2**
  - return the type of the body **E2**.

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( and(E1, E2) , env ) ≜ [ t1 =  typchk ( E1, env)
                                t2 =  typchk ( E2, env)
                                if  (t1 == bool ) and (t2 == bool )
                                    then bool
                                    else TYPEERROR ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( if(E0, E1, E2) , env ) ≜ [ t0 =  typchk ( E0, env)
         if  (t0 != bool ) then TYPEERROR
        else [  t1 =  typchk ( E1, env)
                t2 =  typchk ( E2, env)
                if  (t1 == t2 ) then t2  else TYPEERROR
       ] ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

```
typchk( new (E) , env ) ≜
                    [ t = typchk( E, env );
                      if (t == TYPEERROR)
                      then t
                      else ref ( t ) ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

```
typchk( deref(E) , env ) ≜
                        [ t = typchk( E, env );
                          if (t == ref (tr )
                                 then  tr
                                 else TYPEERROR  ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

```
typchk( assign(E1, E2) , env ) ≜
                         [ t1 = typchk( E1, env );
                           t2 = typchk( E2, env );
                           if (t1 == ref ( t2 ))
                                   then t2
                                   else TYPEERROR ; ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

```
typchk( seq(E1, E2) , env ) ≜
        [ t1 = typchk( E1, env );
          if ( t1 == TYPEERROR ) then TYPEERROR
          else [ t2 = typchk( E2, env );
                 return t2 ] ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

```
typchk( if(E1, E2, E3) , env ) ≜
        [ t1 = typchk( E1, env );
          if ( t1 != bool ) then TYPEERROR
          else [ t2 = typchk( E2, env );
                 t3 = typchk( E3, env );
                 if (t2 != t3)
                    or (t2 == TYPEERROR) or (t3 == TYPEERROR)
                 then TYPEERROR
                 else t2 ] ]
```

Sematics of CALCS (recall)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$\mathit{eval}: \mathrm{AST} \times \mathrm{ENV} \times \mathrm{MEM} \rightarrow \mathrm{VAL} \times \mathrm{MEM}$$

eval(**seq**(E1, E2) , env , m0) ≜ **[**(v1 , m1) = **eval**(E1, env, m0);
eval(E2, env, m1)
]

eval(**if**(E1, E2, E3) , env , m0) ≜
[(v1 , m1) = **eval**(E1, env, m0);
if (v1 = true) **then eval**(E2, env, m1);
else eval(E3, env, m1);
]

CALCS Typing Rules

- Let's compare this with the case for **if** in Map **eval**.

  Here, **both** branches E2 e E3 are analysed;
  and we impose (as a conditions) *t2* == *t3* . [Why?]

```
typchk( if(E1, E2, E3) , env ) ≜
                    [ t1 = typchk( E1, env );
                      if ( t1 != bool ) then TYPEERROR
                      else [ t2 = typchk( E2, env );
                             t3 = typchk( E3, env );
                             if (t2 != t3)
                                or (t2 == TYPEERROR) or (t3 == TYPEERROR)
                             then TYPEERROR
                             else t2 ] ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

```
typchk( while(E1, E2) , env ) ≜
        [ t1 = typchk( E1, env );
          if ( t1 != bool) then TYPEERROR
          else [ t2 = typchk( E2, env );
                 if ( t2 == TYPEERROR) TYPEERROR
                 else bool
               ]
        ]
```

Sematics of CALCS (recall)

Algorithm eval( ) that computes the denotation (value) of any **open** CALCS expression:

$$\mathit{eval}: \mathrm{AST} \times \mathrm{ENV} \times \mathrm{MEM} \rightarrow \mathrm{VAL} \times \mathrm{MEM}$$

```
eval( while(E1, E2) , env , m0) ≜
                    [(v1 , m1) = eval( E1, env, m0 );
                     if (v1 = T) then  [ (v2 , m2) = eval( E2, env, m1);
                           (v,m1) =eval( while(E1, E2), m2) ]
                      else (F , m1) ]
```

CALCS Typing Rules

- Let's compare this with the case for **while** in Map **eval**.

  Here, **body** E2 is analysed exactly once (no iteration); and the returned type is bool . [Why?]

```
typchk( while(E1, E2) , env ) ≜
                    [ t1 = typchk( E1, env );
                      if ( t1 != bool) then TYPEERROR
                      else [ t2 = typchk( E2, env );
                             if ( t2 == TYPEERROR) TYPEERROR
                             else bool
                           ]
                    ]
```

CALCS Typing Rules

- Map **typchk** that computes a type for any CALCS program:

  typechk : CALCS × ENV<TYPE> → TYPE U { TYPEERROR }

  ENV : ID → TYPE

```
typchk( def(s, E1, E2) , env ) ≜
                    [ t1 = typchk( E1, env);
                      if ( t1 == TYPEERROR) then TYPEERROR
                      else envlocal = env.BeginScope();
                      envlocal.assoc(s, t1);
                      t2 = typchk(E2, envlocal);
                      env = envlocal.EndScope();
                      t2 end ]
```

Decidability and Soundness of Typing

- Notice that for any program P and environment *env* which covers all free identifiers of P the typechecking operation

$$\textbf{typchk}(P, \mathit{env})$$

is well defined and always terninates [Why?]

- We may also prove the following result, which related well typing with program evaluation

Theorem: For any CALCS program P and type $\mathcal{T}$,
If **typchk**(P,∅) = $\mathcal{T}$ and **eval**(P, ∅, ∅) = v then v : $\mathcal{T}$.

Decidability and Soundness of Typing

Theorem: For any CALCS program P and type $\mathcal{T}$,
If **typchk**(P,∅) = $\mathcal{T}$ and **eval**(P, ∅, ∅) = v then v : $\mathcal{T}$.

That is, if **typchk**(P,∅) = $\mathcal{T}$ then:

either P does not terminate (loops for ever),

or P terminates and does not get into any runtime errors due to illegal operations.

"*Well-typed programs don´t go wrong*" (Robin Milner)

Decidability and Soundness of Typing

Theorem: For any CALCS program P and type $\mathcal{T}$,
If **typchk**(P,∅) = $\mathcal{T}$ and **eval**(P, ∅, ∅) = v then v : $\mathcal{T}$.

The property stated in the Theorem above is known as "Type Safety". It implies that

"*Well-typed programs don´t go wrong*" (Robin Milner)

NOTE (incompleteness of type checking)
There are many programs P such that eval(P, ∅, ∅) = v but typchk(P, ∅) = **TYPEERROR** ! [Can you give an example?]

Robin Milner

ACM Turing Award (1991)

For three distinct and complete achievements:

1) LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2) ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3) CCS, a general theory of concurrency. In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

Quiz

1. Is the program below well-typed? if not, what is a corrected version.
2. what is the typing environment at `w := ...` ?
3. what is the output of the (possibly corrected) program ?

```
def
        x = 10
        y = new(0)
in
  def
        z = new(y)
        w = new(false)
  in
        while !w do
           w := ((!z := !!z + !y + 1) < x)
        end;
       println !y
  end
end
```

Quiz

1. Is the program below well-typed? if not, what is a corrected version.
2. what is the typing environment at `w := ...` ?
3. what is the output of the (possibly corrected) program ?

```
def
        x = 10              x -> INT
        y = new(0)          y -> REF[INT]
in
  def
        z = new(y)
        w = new(false)
  in
        while !w do
          w := ((!z := !!z + !y + 1) < x)
        end;
       println !y
  end
end
```

Quiz

1. Is the program below well-typed? if not, what is a corrected version.
2. what is the typing environment at `w := ...` ?
3. what is the output of the (possibly corrected) program ?

```
def
        x = 10              x -> INT
        y = new(0)          y -> REF[INT]
in
  def
        z = new(y)          z -> REF[REF[INT]]
        w = new(false)      w -> REF[BOOL]
  in
        while !w do         % !w : BOOL
           w := ((!z := !!z + !y + 1) < x)
        end;
       println !y
  end
end
```