Chapter 7. Types
(Essentials of Programming Languages)

Kwanghoon Choi

Software Languages and Systems Laboratory
Chonnam National University

Outline

This chapter discusses how to use the same technology as for building interpreters to analyze or predict the behavior of programs without running them through type analysis.

- ♢ Values and Their Types
- ♢ Assigning a Type to an Expression
- ♢ CHECKED: A Type-Checked Language
- ♢ INFERRED: A Language with Type Inference

Introduction

Our goal is to analyze a program to predict whether evaluation of a program is safe without certain kinds of errors.

In LETREC, an evaluation is *safe* if and only if

- For every evaluation of a variable, the variable is bound.
- For every evaluation of (diff-exp exp1 exp2), the values of exp1 and exp2 are both num-vals.
- For every evaluation of (zero?-exp exp1), the value of exp1 is a num-val.
- For every evaluation of (if-exp exp1 exp2 exp3), the value of exp1 is a bool-val.
- For every evaluation of (call-exp rator rand), the value of rator is a proc-val.

Introduction (Cont.)

These conditions assert that each operator is performed only on operands of the correct type.

Violations of these conditions are called *type errors*.

A safe evaluation may still fail by divide-by-zero, taking the car of an empty list, nontermination, etc.

- Predicting safety for these conditions is much harder than guaranteeing the conditions in the previous slide

Introduction (Cont.)

Our goal is to write a procedure that looks at the program text and either accepts or rejects it. (cf. The soundness of a type analysis)

```
if 3 then 88 else 99
proc (x) (3 x)
proc (x) (x 3)
proc (f) proc (x) (f x)
let x = 4 in (x 3)
(proc (x) (x 3) 4)
let x = zero?(0) in -(3, x)
(proc (x) -(3,x) zero?(0))
let f = 3 in proc (x) (f x)
(proc (f) proc (x) (f x) 3)
letrec f(x) = (f -(x,-1)) in (f 1)
```

7.1 Values and Their Types

Since the safety conditions talk only about num-val, bool-val, and proc-val, one might think that it would be enough to keep track of these three types. But that is not enough.

- e.g., (f 1)

A finer information about procedures is needed. Therefore, the type structure of LETREC is:

Type ::= int $\boxed{\text{int-type ()}}$

Type ::= bool $\boxed{\text{bool-type ()}}$

Type ::= (Type $\rightarrow$ Type) $\boxed{\text{proc-type (arg-type result-type)}}$

See examples of values and their types in the next slide.

Q. Explain differences between the two kinds of function type, proc-val and Type $\rightarrow$ Type.

7.1 Values and Their Types (Cont.)

Examples: The value of an expression exp has type ty.

	an expression	has type
	3	int
	-(33, 22)	int
	zero? (11)	bool
	proc (x) - (x,11)	(int $\rightarrow$ int)
The	proc (x) let y $=$ -(x,11) in -(x,y)	(int $\rightarrow$ int)
value	proc (x) if x then 11 else 22	(bool $\rightarrow$ int)
of	proc (x) if x then 11 else zero? (11)	no type
	proc (x) proc (y) if y then x else 11	(int $\rightarrow$ (bool $\rightarrow$ int))
	proc (f) (f 3)	((int $\rightarrow$ t) $\rightarrow$ t)
	proc (f) proc (x) (f (f x))	((t $\rightarrow$ t) $\rightarrow$ (t $\rightarrow$ t))
		for any type t

7.1 Values and Their Types (Cont.)

"*An expressed value v is of type t*" is defined by induction on t:

- v is of type int if and only if v is a num-val.
- v is of type bool if and only if v is a bool-val.
- v is of type (t1 $\rightarrow$ t2) if and only if v is a proc-val with the property that if it is given an argument of type t1, then one of the following things happens:
  1. it returns a value of type t2
  2. it fails to terminate
  3. it fails with an error other than a type error.

"v has type t" instead of "v is of type t"

Some value can be of more than one type, and some value can have no type.

7.1 Values and Their Types (Cont.)

"*An environment env is of type tenv*" is defined as

- env=[x1=val1, ..., xn=valn]
- tenv=[x1=t1, ..., xn=tn]
- val1 is of type t1, ..., valn is of type tn.

Then env is said to *respect* tenv.

7.2 Assigning a Type to an Expression

In order to analyze programs, we write a procedure *type-of* that takes an expression and predicts the type of its value.

- (type-of exp tenv) $=$ t
- tenv: a type environment mapping each variable to a type
- t: a type assigned to the expression with the property that whenever exp is evaluated in an environment respecting tenv, one of the following happens
  - the resulting value v has type t
  - the evaluation does not terminate, or
  - the evaluation fails on an error other than a type error.

Well-typed expression, ill-typed expression

7.2 Assigning a Type to an Expression (Cont.)

Simple typing rules

(type-of (const-exp num) tenv) = int

(type-of (var-exp var) tenv) = tenv (var)

$$\frac{\text{(type-of exp1 tenv)} = \text{int}}{\text{(type-of (zero?-exp exp1) tenv)} = \text{bool}}$$

$$\frac{\text{(type-of exp1 tenv)} = \text{int} \quad \text{(type-of exp2 tenv)} = \text{int}}{\text{(type-of (diff-exp exp1 exp2) tenv)} = \text{int}}$$

$$\frac{\text{(type-of exp1 tenv)} = \text{t1} \quad \text{(type-of body [var=t1]tenv)} = \text{t2}}{\text{(type-of (let-exp var exp1 body) tenv)} = \text{t2}}$$

7.2 Assigning a Type to an Expression (Cont.)

Simple typing rules (Cont.)

$$\frac{\begin{array}{c}\text{(type-of exp1 tenv)} = \text{bool} \\ \text{(type-of exp2 tenv)} = \text{t} \\ \text{(type-of exp3 tenv)} = \text{t}\end{array}}{\text{(type-of (if-exp exp1 exp2 exp3) tenv)} = \text{t}}$$

$$\frac{\text{(type-of rator tenv)} = (\text{t1}\rightarrow\text{t2}) \qquad \text{(type-of rand tenv)} = \text{t1}}{\text{(type-of (call-exp rator rand) tenv)} = \text{t2}}$$

$$\frac{\text{(type-of body [var=t1]tenv)} = \text{t2}}{\text{(type-of (proc-exp var body) tenv)} = (\text{t1}\rightarrow\text{t2})}$$

7.2 Assigning a Type to an Expression (Cont.)

How can we find the type t1 in the typing rule for procedure expression?

$$\frac{\text{(type-of body [var=t1]tenv)} = \text{t2}}{\text{(type-of (proc-exp var body) tenv)} = (\text{t1}\rightarrow\text{t2})}$$

Two standard designs for resolving this problem are type checking and type inference.

- Type checking: Programmers supply the missing type information
- Type inference: Type checkers attempt to infer the missing type information based on how the variables are used in the program.

7.3 CHECKED: A Type-Checked Language

CHECKED = LETREC + the programmer's annotation of the types of all bound variables

Examples

```
proc (x : int) -(x,1)

letrec
 int double (x : int) =
        if zero? (x) then 0
        else -((double -(x,1)), -2)
in double

proc (f : (bool->int))
  proc (n : int)
    (f zero? (n))
```

7.3 CHECKED: A Type-Checked Language (Cont.)

The syntax

Expression ::= proc (Identifier : Type) Expression $\boxed{\text{proc-exp (var ty body)}}$

Expression ::=

letrec Type Identifier (Identifier : Type) =

Expression in Expression

$\boxed{\text{letrec-exp (p-result-type p-name b-var b-var-type p-body letrec-body)}}$

7.3 CHECKED: A Type-Checked Language (Cont.)

Typing "proc (var : t1) body":

$$\frac{\text{(type-of body [var=t1]tenv) = t2}}{\text{(type-of (proc-exp var t1 body) tenv) = (t1}\rightarrow\text{t2)}}$$

Typing "letrec t2 p (var : t1) procbody in letrecbody"

$$\frac{\begin{array}{c}\text{(type-of procbody [var=t1][p=t1}\rightarrow\text{t2]tenv) = t2} \\ \text{(type-of letrecbody [p=t1}\rightarrow\text{t2]tenv]) = t}\end{array}}{\text{(type-of (letrec-exp t2 p var t1 procbody lterecbody) tenv) = t}}$$

7.3 CHECKED: A Type-Checked Language (Cont.)

An implementation of type-of-program and type-of in Figure 7.1, 7.2, 7.3. with auxiliary functions

- check-equal-type!
- report-unequal-type
- type-to-external-form

7.4 INFERRED: A Language with Type Inference

Writing down the types in the program is helpful for documentation, but it can be time-consuming.

Another design is to have the compiler figure out the types of all the variables, based on

- how the variables are used, and
- utilizing any hints the programmers might give

$\Rightarrow$ Type inference

7.4 INFERRED: A Lang. with Type Inf. (Cont.)

Examples

```
letrec
 ? foo (x : ?) = if zero? (x)
        then 1
        else -(x, (foo -(x,1)))
in foo

letrec
 ? even (x : int) =
      if zero?(x) then 1 else (odd -(x,1))
 bool odd (x : >) =
     if zero?(x) then 0 else (even -(x,1))
in (odd 13)
```

7.4 INFERRED: A Lang. with Type Inf. (Cont.)

To specify this syntax of optional type, a new grammar is:

- Optional-type ::= ? $\boxed{\text{no-type ()}}$
- Optional-type ::= Type $\boxed{\text{a-type (ty)}}$
- Expression ::= proc (Identifier : Optional-type) Expression $\boxed{\text{proc-exp (var otype body)}}$
- Expression ::= letrec Optional-type Identifier (Identifier : Optional-type) = Expression in Expression $\boxed{\text{letrec-exp (p-result-otype p-name b-var b-var-otype p-body letrec-body)}}$

7.4 INFERRED: A Lang. with Type Inf. (Cont.)

The omitted types are treated as unknowns that we need to find.

We traverse the abstract syntax tree and generate equations between these types, possibly including these unknowns.

We then solve the equations for the unknown types.

7.4 INFERRED: A Lang. with Type Inf. (Cont.)

In other words, to infer the type of an expression,

- we'll introduce a type variable for every subexpression and every bound variable,
- generate the constraints for each subexperssion, and then
- solve the resulting equations.

7.4 INFERRED: A Lang. with Type Inf. (Cont.)

For each type rule, there are some equations that must hold between the types of subexpressions.

$$
\begin{array}{lcl}
(diff - exp\ e_1\ e_2) & : & t_{e_1} = int,\ t_{e_2} = int,\ t_{(diff-exp\ e_1\ e_2)} = int \\
(zero? - exp\ e_1) & : & t_{e_1} = int,\ t_{zero?-exp\ e_1} = bool \\
(proc - exp\ var\ body) & : & t_{(proc-exp\ var\ body)} = (t_{var} \rightarrow t_{body}) \\
(call - exp\ rator\ rand) & : & t_{rator} = (t_{rand} \rightarrow t_{(call-exp\ rator\ rand)})
\end{array}
$$

7.4 INFERRED: A Lang. with Type Inf. (Cont.)

Examples for Type Inference:

- proc (f) proc (x) -((f 3), (f x))
  - substitution
- proc (f) (f 11)
  - polymorphism
- if x then -(x,1) else 0
  - inconsistent types
- proc (f) zero? ((f f))
  - occurrence check

7.4.1 Substitutions

A substitution is a set of equations where the left-hand sides are all variables.

Type Expressions

Type ::= %tvar-type Number $\boxed{\text{tvar-type (serial-number)}}$

Basic operations on type expressions

- (apply-one-subst ty0 tvar ty1): the type obtained by substituting ty1 for every occurrence of tvar in ty0
- (apply-subst-to-type ty subst): the type obtained by replacing each type variable by its binding in the substitution

Constructors for substitutions

- (empty-subst), (extend-subst subst tvar ty)

7.4.2 The Unifier

(unifier t1 t2 subst) returns another substitution subst', which is the smallest extension of subst satisfying (subst' t1 = subst' t2).

- It applies subst to ty1 and ty2 resulting in ty1' and ty2'.
- If ty1' is equal to ty2', it returns subst.
- If ty1' is a type variable that does not occur in ty2', it returns (extend-subst subst ty1 ty2)
- If ty2' is a type variable that does not occur in ty1', it returns (extend-subst subst ty2 ty1)
- If neither ty1' nor ty2' is a type variable, we analyze the substructure of these types further.
  - If both of ty1' and ty2 are int or bool, return subst.
  - If ty1'=ty1'' $\rightarrow$ ty1''', ty2'= ty2'' $\rightarrow$ ty2''', (unifier ty1'' ty2'' (unifier ty1''' ty2''' subst))

7.4.3 Finding the Type of an Expression

We convert optional types to types with unknowns by defining a fresh type variable for each ? using otype->type.

(type-of exp tenv subst) = (ty, subst')
  : Figure 7.6, 7.7, 7.8, and 7.9

For testing type inference, a way of comparison of two types in external form is needed.
  : e.g., tvar1 ->tvar1 is considered equal to tvar2 ->tvar2