Lecture 20: Types and Type Systems

CS4400 Programming Languages

Introduction

> *This part is partially based on notes by Norman Ramsey and on the book Types and Programming Languages by Benjamin Pierce*

What is a type?

- In a Haskell-like language (simplified by ignoring type classes)

```haskell
1 + 5 :: Integer

"hello " ++ "world" :: String

n > 10 :: Bool -- if n :: Integer

if n > 10 then "Yes" else "No" :: String  -- if n :: Integer

\x -> x + 10 :: Integer -> Integer
```

- Types classify program phrases according to the kinds of values they compute
- They are predictions about values
- Static *approximation* of runtime behavior – conservative

Why types?

- Static analysis: detect (potential) runtime errors before code is run:

```haskell
1 + True

10 "hello" -- application of the number 10 to a string?
```

- E.g., Python is happy to accept the following function definition:

```
def f(x):
  if x < 10:
    x(x, 10)
  else:
    "Hello " + x
```

  What happens at runtime, when the function is called as `f(4)`?

- Enforcing access policy (private/protected/public methods)
- Guiding implementation (type-based programming)
- Documentation: types tell us a lot about functions and provide a documentation that is repeatedly checked by the compiler (unlike comments)
- Help compilers choose (more/most) efficient runtime value representations and operations
- Maintenance: if we change a function's type, the type checker will direct us to all use sites that need adjusting

What is a type system?

> A tractable syntactic method for proving the absence of *certain* program behavior.

- They are studied on their own as a branch of mathematics/logic: type theory
- Original motivation: avoiding Russell's paradox
- Type systems are generally *conservative*:

```
1 + (if True then 10 else "hello")
```

  would behave OK at runtime, but is, nevertheless, typically rejected by a static type checker (e.g., Haskell's)

What kind of errors are typically not detected by type systems?

- Division by zero
- Selecting the head of an empty list
- Out-of-bounds array access
- Non-termination

Consideration: a program which mostly runs numeric computations will benefit from a strong static type system less than a program which transforms various data structures

Terminology: type systems

Dynamic types are checked at runtime, typically when operations are performed on values

Static types are checked before program is (compiled and) run

What is a type safe language?

- Can a dynamically typed language be safe?
- Is a statically typed language automatically type safe?

Dynamic:

- Consider Python

  ```
  >>> 1 + "hello"
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: unsupported operand type(s) for +: 'int' and 'str'

  >>> "hello" + 1
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: can only concatenate str (not "int") to str
  ```

- Rejects applying + to incompatible arguments – protects from behavior that is incompatible with the abstractions of integers and strings

Static:

- Consider C:

  ```c
  1 int array[] = {1, 2, 3, 4};
  2 char string[] = "Hello, world!";
  3
  4 printf("%d %d %d %d\n", array[0], array[1], array[2], array[3]);
  5
  6 array + 2;
  7 string + 3;
  ```

  Here the compiler will merely complain about unused values on lines 6 and 7. But what does the expression `array + 2` mean? Of course, a C programmer knows, that arrays are just pointers, and so adding two to a pointer merely shifts where the pointer points to. But is it complatible with the abstraction of array?

So, again: what is a type safe-language?

> A (type-)safe language is one that protects its own abstractions.

This means that a safe language won't allow a programmer to apply operations which are not "sensible" for the given arguments, potentially leading to unexpected/undefined behavior at runtime.

Specifying Type Systems

Abstract syntax + Types + Typing rules (+ auxiliary operations)

- A type system is typically specified as a set of rules which allow assigning types to abstract syntax trees – similarly to how evaluators assign values to abstract syntax trees
- The goal: determine what type an expression (program phrase) has – if it has one
- A type judgement – mathematically:

$$\vdash e : t$$

Read: "e has type t"

- A "type judgement" – in Haskell

```
typeOf e = t
```

- Typing rules tell us how to arrive at the above conclusion for a particular expression
- This is based on syntax – in other words type checking (and typing rules) are, typically, **syntax-directed**
- On paper, typing rules are usually expressed as inference rules:

$$\frac{\text{1st premise} \quad \text{2nd premise} \quad \text{3rd premise ...}}{\text{conclusion}}$$

- Such a rule can be read as "If 1st premise AND 2nd premise AND 3rd premise ... are all true, then the conclusion is also true"
- If we can show that the premises hold, the rule allows us to conclude that is below the line
- If a rule has no premises, it is an axiom
- Here are some examples, written mathematically

$$\frac{}{\vdash 3 : \text{Integer}}$$

"The type of the value 3 is Integer"

$$\frac{n \text{ is an integer value}}{\vdash n : \text{Integer}}$$

"If n is an integer value, then the type of n is Integer"

$$\frac{\vdash e_1 : \text{Integer} \qquad \vdash e_2 : \text{Integer}}{\vdash e_1 + e_2 : \text{Integer}}$$

"If the type of e_1 is Integer and the type of e_2 is Integer, then the type of expression $e_1 + e_2$ is also Integer"

- We can (and will) view these inference rules as a fancy way of writing Haskell functions
- Let us first define datatypes for expressions (for now only integer numbers and addition) and types (only integers)

```
data Expr = Num Integer
          | Add Expr Expr

data Type = TyInt
```

- The above two rules as a Haskell type checker:

```
typeOf :: Expr -> Maybe Type   -- an expression might not have a type
typeOf (Num n) = return TyInt
typeOf (Add e1 e2) =
  do TyInt <- typeOf e1
     TyInt <- typeOf e2
     return TyInt
```

- Note that the return type of `typeOf` is `Maybe Type`
  - This is to allow for the possibility that an expression might not have a type (although in this trivial language, all expressions do)
- We use the **do** notation (together with `return`) to simplify the definition. The above is equivalent to:

```
typeOf :: Expr -> Maybe Type   -- an expression might not have a type
typeOf (Num n) = Just TyInt
typeOf (Add e1 e2) =
  case typeOf e1 of
       Just TyInt -> case typeOf e2 of
                          Just TyInt -> Just TyInt
                          _ -> Nothing
       _ -> Nothing
```

- To make the connection a little more explicit, we will write inference rules as a mix of Haskell and math:

```
------------------
 |- Num n : TyInt


 |- e1 : TyInt    |- e2 : TyInt
--------------------------------
 |- Add e1 e2 : TyInt
```

- More typing rules (we add a few new expression shapes + a new type for booleans):

```haskell
data Expr = ...
          | Bool Bool
          | And Expr Expr
          | Not Expr
          | Leq Expr
          | If Expr Expr Expr

data Type = ...
          | TyBool
```

```
--------------------
 |- Bool b : TyBool


 |- e1 : TyBool    |- e2 : TyBool
---------------------------------
 |- And e1 e2 : TyBool


 |- e : TyBool
-------------------
 |- Not e : TyBool


 |- e1 : TyInt   |- e2 : TyInt
-------------------------------
 |- Leq e1 e2 : TyBool


 |- e1 : TyBool    |- e2 : t    |- e3 : t
----------------------------------------
 |- If e1 e2 e3 : t
```

How do we apply these rules?

- We build derivations!
- But what are derivations?
- A derivation is a (proof) tree built by *consistently* replacing variables in inference rules by concrete terms
- At the bottom of the tree is the typing judgment we are trying to show

Examples:

1. A numeric literal

   ```
   ------------------
    |- Num 3 : TyInt
   ```

   Nothing else needed here, since the rule is an axiom and doesn't have any conditions (premises)

2. Addition of two numbers

   ```
    |- Num 3 : TyInt   |- Num 3 : TyInt
   -------------------------------------
    |- Add (Num 3) (Num 4) : TyInt
   ```

3. Boolean expression:

   ```
    |- Bool True : TyBool          |- Bool False : TyBool  |- Bool True : TyBo
   -----------------------------   ---------------------------------------------
    |- Not (Bool True) : TyBool     |- And (Bool False) (Bool True) : TyBool
   ---------------------------------------------------------------------------
    |- And (Not (Bool True)) (And (Bool False) (Bool True)) : TyBool
   ```

   Prettier:

   $$\dfrac{\dfrac{\vdash \text{Bool True} : \text{TyBool}}{\vdash \text{Not (Bool True)} : \text{TyBool}} \quad \dfrac{\vdash \text{Bool False} : \text{TyBool} \quad \vdash \text{Bool True} : \text{TyBool}}{\vdash \text{And (Bool False) (Bool True)} : \text{TyBool}}}{\vdash \text{And (Not (Bool True)) (And (Bool False) (Bool True))} : \text{TyBool}}$$

4. Conditional involving booleans and integers

```
 |- Num 3 : TyInt   Num 4 : TyInt
----------------------------------
 |- Leq (Num 3) (Num 4) : TyBool
--------------------------------------
 |- Not (Leq (Num 3) (Num4)) : TyBool    |- Num 3 : TyInt    |- Num 5 : TyInt
--------------------------------------------------------------------------------
 |- If (Not (Leq (Num 3) (Num 4))) (Num 3) (Num 5)
```

5. A failing one:

```
 |- Bool True : TyBool  |- Num 3 : TyInt
-----------------------------------------
 |- Add (Bool True) (Num 3) : ?
```

   We have no rule to apply here. We would need `Num 3` to have type `TyBool` and there is no rule that allows us to derive this. Hence, the above expression cannot be type-checked.

Type-checking Involving Variables

Syntax extensions:

```haskell
data Expr = ...
          | Var Variable
          | Let Variable Expr Expr
```

How do we deal with variables?

- We need to keep track of types assigned to variables
- Idea: Like for an (environment-based) evaluator for expressions, use an environment
- The environment maps variables *to types*

```
type TyEnv = Map Variable Type
```

Example rules:

```
  t <- get x tenv
-------------------
 tenv |- Var x : t
```

```
 tenv |- e1 : t1    add x t1 env |- e2 : t2
--------------------------------------------
         tenv |- Let x e1 e2 : t2
```

In Haskell:

```haskell
typeOf :: TyEnv -> Expr -> Maybe Type
...
typeOf tenv (Add e1 e2) =     -- previous cases need to be refactored to use tenv
  do TyInt <- typeOf tenv e1
     TyInt <- typeOf tenv e2
     return TyInt
...
typeOf tenv (Var x) = get tenv x      -- NEW: variable lookup
typeOf tenv (Let x e1 e2) =           -- NEW: let-binding
  do t1 <- typeOf tenv e1             -- get type of e1
     t2 <- typeOf (add x t1 tenv) e2  -- get the type of e2, assuming x : t1
     return t2
```