Lecture 23: Type-Checking

CSCI 101
Spring, 2019

Kim Bruce

What does it mean for a language to be type-safe?

Safe Languages

- Two kinds of execution errors
  - Trapped errors: cause computation to halt immediately.
    - Divide by zero, null pointer exception
  - Untrapped errors: go unnoticed and later cause problems.
    - Access an illegal address, e.g., array bounds error.
- Program fragment is *safe* if it causes no untrapped errors.
  - Language is safe if all program fragments are safe.

See "Type Systems" by Luca Cardelli
http://lucacardelli.name/Papers/TypeSystems%201st%20Edition.US.pdf

Strongly Typed Languages

- Language designates *forbidden* errors
  - those that are not allowed to happen.
  - should include all untrapped errors
- Program fragment is *well behaved* if it generates no forbidden errors.
- Language where all legal programs are well behaved is *strongly typed*

Static vs. Dynamic Typing

- Most use static typing
  - including C/Java/ML/Haskell
  - binding of types to variables done at translation time.
  - Find errors earlier, but conservative.
- dynamic typing
  - LISP/Scheme/Racket/Python/Javascript/Grace
  - binding of type to value, not variable.
    - thus binding of type to variable changes dynamically
  - Dynamic more flexible, but more overhead.

(Static) Type Checking

Static Type Checking

- Static type-checkers for strongly-typed languages (i.e., rule out all "bad" programs) must be conservative:
  - Rule out some programs without errors.
- if (*program-that-could-run-forever*) {
  expression-w-type-error;
  } else {
  expression-w-type-error;
  }

Type checking

- Most statically typed languages also include some dynamic checks.
  - array bounds.
  - Java's instanceof
  - typecase or type casts
- Pascal statically typed, but not strongly typed
  - variant records (*essentially union types)*, dangling pointers
- Haskell, ML, Java strongly typed
- C, C++ not strongly typed

Type Compatibility

- When is x := y legal?

```
Type T = Array [1..10] of Integer;
Var A, B : Array [1..10] of Integer;
C : Array [1..10] of Integer;
D : T;
E : T;
```

- Name Equivalence$_A$ (*Ada)*
- Name Equivalence (*Pascal, Modula-2, Java)*
- Structural Equivalence (*Modula-3, Java arrays only, Grace)*

Structural Equivalence

- Can be subtle:

```
T1 = record a : integer; b : real end;
T2 = record c : integer; d : real end;
T3 = record b : real; a : integer end;
```

- Which are the same?

```
T = record info : integer; next : ^T end;
U = record info : integer; next : ^V end;
V = record info : integer; next : ^U end;
```

Type Checking & Inference

- Write explicit rules. Let a, b be expressions
  - if a, b:: Integer,
    then a+b, a*b, a div b, a mod b:: Integer
  - if a, b:: Integer then a < b, a = b, a > b : Bool
  - if a, b: Bool then a && b, a || b: Bool
  - ...

Formal Type-Checking Rules

- Can rewrite more formally.
- Expression may involve variables, so type check wrt assignment E of types to variables.
  - E.g., E(x) = Integer, E(b) = Bool, ...

Hypothesis → $E(x) = t$

$$\frac{E(x) = t}{E \vdash x : t}$$

$E \vdash x : t$ ← *Conclusion*

$$\frac{E \vdash a : int,\ E \vdash b : int}{E \vdash a+b : int}$$

Can write formally

Function Application:

$$\frac{E \vdash f: \sigma \to \tau, \quad E \vdash M : \sigma}{E \vdash f(M) : \tau}$$

Function Definition:

$$\frac{E \cup \{v:\sigma\} \vdash \text{Block} : \tau}{E \vdash \text{fun } (v:\sigma) \text{ Block} : \sigma \to \tau}$$

Can write for all language constructs.
Based on context free grammar.
Can read off type-checking algorithm.

Haskell Type Inference

How does Haskell know what you meant?

Haskell Type Inference

1. An identifier should be assigned the same type throughout its scope.
2. In an "if-then-else" expression, the condition must have type Bool and the "then" and "else" portions must have the same type. The type of the expression is the type of the "then" and "else" portions.
3. A user-defined function has type a → b, where a is the type of the function's parameter and b is the type of its result.
4. In a function application of the form f x, there must be types a and b such that f has type a → b, x has type a, and the application itself has type b.

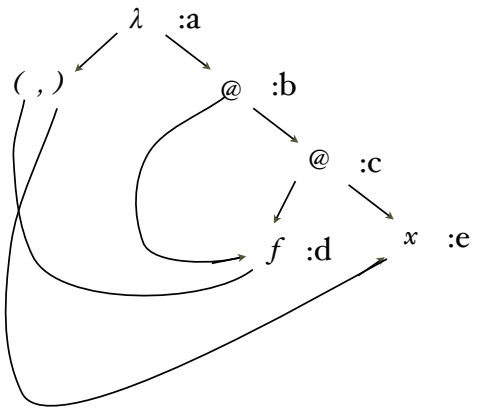
Examples of Type Inference

- Use rules to deduce types:

```
map = \ f -> \ l ->
          if l == [] then []
              else (f (head l)): (map f (tail l))
```

- map:: a → b *because function*
- f :: a, `\ l -> ...` :: b, Thus b = c → d
- l :: c, `if l = [] then ...` :: d
- `...`

@ = application




```
double(f,x) = f(f(x))
```

or equivalently

```
double = \ (f,x) -> f(f(x))
```

Outcome of Type Inference

- Overconstrained: no solution

```
Prelude> tail 7

<interactive>:1:5:
    No instance for (Num [a])
      arising from the literal `7' at <interactive>:1:5
    Possible fix: add an instance declaration for (Num [a])
    In the first argument of `tail', namely `7'
    In the expression: tail 7
    In the definition of `it': it = tail 7
```

- Underconstrained: polymorphic
- Uniquely determined

By the way, ...

- Inference due to Hindley-Milner
- SML/Haskell type inference is doubly exponential in the worst case!
- Can write down terms t_n of length n such that the length of the type of t_n is of length 2^{2^n}
- Luckily, it doesn't matter in practice,
  - no one writes terms whose type is exponential in the length of the term!

Restrictions on ML/Haskell Polymorphism

- Type $(a \to b) \to ([a] \to [b])$ stands for:
  - $\forall a.\ \forall b.\ (a \to b) \to ([a] \to [b])$
- Haskell functions may not take polymorphic arguments. E.g., no type:
  - $\forall b.\ ((\forall a.(a \to a)) \to (b \to b))$
  - define: foo f (x,y) = (f x, f y)
  - id z = z
  - foo id (7, True) -- gives type error!
  - Type of foo is only (t -> s) -> (t, t) -> (s, s)

Restrictions on Implicit Polymorphism

Polymorphic types can be defined at top level or in let clauses, but can't be used as arguments of functions

```
id x = x
   in (id "ab", id 17)
```

OK, but can't write

```
test g = (g "ab", g 17)
```

Can't find type of test w/unification.
More general type inference is undecidable.

Explicit Polymorphism

Easy to type w/ explicit polymorphism:

```
test (g: forall t.t -> t) = (g "ab", g 17)
   in test (\t => \(x:t) -> x)
```

Languages w/explicit polymorphism:
Clu, Ada, C++, Eiffel, Java 5, C#, Scala, Grace

Explicit Polymorphism

- Clu, Ada, C++, Java
- C++ macro expanded at link time rather than compile time.
- Java compiles away polymorphism, but checks it statically.
- Better implementations keep track of type parameters.

Summary

- Modern tendency: strengthen typing & avoid implicit holes, but leave explicit escapes
- Push errors closer to compile time by:
  - Require over-specification of types
  - Distinguishing between different uses of same type
  - Mandate constructs that eliminate type holes
  - Minimizing or eliminating explicit pointers
- Holy grail: Provide type safety, increase flexibility