Type Systems

COMP 311
Rice University
Houston, Texas

Type Systems

Type Systems for Programming Language were invented by mathematicians before electronic computers were invented.

What is a type? A meaningful subset of the set of the domain of data values used in a program. Types are used to describe the intended behavior of program operations.

The concept is derived from mathematics; functions have specified domain and co-domains sets which are often called types. In fact, a function with domain *A* and co-domain *B* is often said to have type $A \to B$. Moveover, the variables used in mathematical formulas typically have specified types

Mathematicians informally check that uses of functions and variables in formulas are respected just as they informally check that reasoning used in proofs is sound.

In computer programs, some forms of “type checking” can be formalized and automated.

Type Systems

Foundation: type systems for functional languages

- Canonical functional language: the □-calculus

  $M ::= c \mid x \mid (M\ M) \mid (\square x\ \ M)$

  $c \,\square\, C$ (a set of constants)

  $x \,\square\, V$ (a set of variables)

  So the □-calculus is really a family of languages that differ only in the constants and variables.

- A simple type system for the □-calculus.
  - A type has the form

    $\square ::= b \mid \square\square\ \ \square$

    $b \,\square\, B$ (a set of base types)

  - Augmented language syntax:

    $M ::= c \mid x \mid (M\ M) \mid (\square x{:}\square\ M)$

Type Systems

Typing rules for the (simply) typed λ-calculus

- Each constant $c \in C$ has a given type $\tau(c)$
- Since an expression *M* inside a λ-calculus program may contain free variables, type rules keep track of the types of these variables using a type environment $\Gamma : V \rightarrow T$ where *T* denotes the set of types τ. This type environment is simply a symbol table that records a symbol's type!
- Assume $M: \sigma \rightarrow \tau$ and $N: \sigma$. Then $(M\ N): \tau$.
- Assume $x: \sigma$ implies that $M: \tau$. Then $(\lambda x: \sigma.\ M): \sigma \rightarrow \tau$

The process of assigning types to λ-calculus programs can be rigorously formalized as a *natural deduction system* where the formal sentences are typing judgements of the form $\Gamma \mid\!- M: \tau$ and the mechanisms for generating *true* sentences are

- *axioms* of the form $\Gamma \mid\!- M: \tau$ and
- *inference rules* of the form

$$\Gamma_1 \mid\!- M_1: \tau_1,\ \ldots,\ \Gamma_N \mid\!- M_N: \tau_N \quad \Rightarrow \quad \Gamma \mid\!- M: \tau$$

Type Systems

Explanation:

- Axioms are typing judgements that are manifestly true.
- Inference rules produce true consequences given true premises
- Common notation for rules

$$\frac{\square_1 \text{ |- } M_1 : \square_1, \ldots, \square_N \text{ |- } M_N : \square_N}{\square \text{ |- } M : \square}$$

Natural deduction rules for (simply typed □-calculus) [□ is arbitrary]

- $\square, x{:}\square \text{ |- } x : \square$
- $\square \text{ |- } c : \square(c)$
- $$\frac{\square \text{ |- } M{:}\square\ \square\ \square,\ \square \text{ |- } N{:}\square}{\square \text{ |- } (M\ N) : \square}$$ (□ elimination or application)
- $$\frac{\square,\ x{:}\square \text{ |- } M : \square}{\square \text{ |- } (\square x{:}\square\ M) : \square\ \square\ \square}$$ (□ introduction or abstraction)

Example λ-calculus language:

- $C = I \cup F$ where

  $I = \{\ldots, -2, -1, 0, 1, 2 \ldots\}$

  $F = \{+, -, *, /\}$

- $V = \{a, b, \ldots z, aa, \ldots\}$
- $B = \{\text{int}\}$
- $\tau(i) = \text{int}$ for $i \in I$; $\tau(f) = \text{int} \rightarrow (\text{int} \rightarrow \text{int})$

- What is type of $(\lambda f: \text{int} \rightarrow \text{int}\ (\lambda g: \text{int} \rightarrow \text{int}\ (\lambda x: \text{int}\ (f\,(g\ x)))))$?

  $(\text{int} \rightarrow \text{int}) \rightarrow ((\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}))$

  which is the “curried” form of

  $(\text{int} \rightarrow \text{int}) \times (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$

Type Systems : Sample Typing Proof

Show:

$$\varnothing \mid (\lambda f\!: \mathbf{int}\to\mathbf{int}\,.\,(\lambda g\!: \mathbf{int}\to\mathbf{int}\,.\,(\lambda x\!:\mathbf{int}\,.\,(f\,(g\,x))))):\ (\mathbf{int}\to\mathbf{int})\to((\mathbf{int}\to\mathbf{int})\to(\mathbf{int}\to\mathbf{int}))$$

Tree1:

$$\frac{f\!:\mathbf{int}\to\mathbf{int},\, g\!:\mathbf{int}\to\mathbf{int},\, x\!:\mathbf{int} \mid g\!:\mathbf{int}\to\mathbf{int}, \qquad f\!:\mathbf{int}\to\mathbf{int},\, g\!:\mathbf{int}\to\mathbf{int},\, x\!:\mathbf{int} \mid x\!:\mathbf{int}}{f\!:\mathbf{int}\to\mathbf{int},\, g\!:\mathbf{int}\to\mathbf{int},\, x\!:\mathbf{int} \mid (g\,x)\!:\mathbf{int}}$$

Tree2:

$$\frac{\dfrac{\dfrac{\dfrac{f\!:\mathbf{int}\to\mathbf{int},\, g\!:\mathbf{int}\to\mathbf{int},\, x\!:\mathbf{int} \mid f\!:\mathbf{int}\to\mathbf{int}, \qquad \text{Tree1}}{f\!:\mathbf{int}\to\mathbf{int},\, g\!:\mathbf{int}\to\mathbf{int},\, x\!:\mathbf{int} \mid (f\,(g\,x))\!:\mathbf{int}}}{f\!:\mathbf{int}\to\mathbf{int},\, g\!:\mathbf{int}\to\mathbf{int} \mid (\lambda x\!:\mathbf{int}\,.\,(f\,(g\,x)))\!:\mathbf{int}\to\mathbf{int}}}{f\!:\mathbf{int}\to\mathbf{int} \mid ((\lambda g\!:\mathbf{int}\to\mathbf{int}\,.\,(\lambda x\!:\mathbf{int}\,.\,(f\,(g\,x)))) : (\mathbf{int}\to\mathbf{int})\to(\mathbf{int}\to\mathbf{int})}}{\varnothing \mid (\lambda f\!:\mathbf{int}\to\mathbf{int}\,.\,(\lambda g\!:\mathbf{int}\to\mathbf{int}\,.\,(\lambda x\!:\mathbf{int}\,.\,(f\,(g\,x))))) : (\mathbf{int}\to\mathbf{int})\to((\mathbf{int}\to\mathbf{int})\to(\mathbf{int}\to\mathbf{int}))}$$

Type Systems

Question: what is the relationship between the (untyped) λ-calculus and the (simply) typed λ-calculus?

- Many expressions (and complete programs) in the (untyped) λ-calculus cannot be typed in the (simply) typed λ-calculus!
  - Example: (λ*x* (*x x*))

    x requires a type $\tau = \tau \rightarrow \tau$, but no such type exists
- If the constant operations terminate for all inputs, then every typable program terminates for all inputs!

Question: is there a tractable algorithm for determining the type of an expression in the (untyped) λ-calculus and rejecting the expression if no such type exists? Yes! The standard algorithm (called the *type reconstruction algorithm*) is based on a very simple idea:

- generate a distinct type variable for every subexpression of the given expression *M,*
- record the equality constraints between these variables dictated by the typing rule that matches the program context in which the subexpression associated with each variable appears, and
- solve these constraints.

Type Systems

More details:

_ **Create only one type variable for each distinct program variable; all occurrences of a given variable must have the same type!**

_ **Create a "compound type variable" □ □ □ for each subexpression that appears as the head *M* of an application (*M N*).**

_ **For each constant *c*, assign it the type □(*c*). Every subexpression now has a symbolic type.**

_ **For each application (*M N*) where M has symbolic type □ □ □, equate □ with the symbolic type of *N* and equate □ with the symbolic type of (*M N*).**

_ **For each abstraction (□*x* *M*) where *x* has symbolic type □ and *M* has symbolic type □, equate □ □ □ with the symbolic type of (□*x* *M*).**

_ **Solve the resulting set of equations on symbolic types (which are just symbolic expressions that can be represented as trees) using the *unification* algorithm (invented by John Alan Robinson, a professor of philosophy at Rice in the 1960's). The generated solution is a substitution mapping type variables to symbolic types.**

Type Systems

What is unification? Tree pattern matching where match variables only appear as leaves.

- A naïve recursive algorithm can be written in a few lines of Scheme or ML.
- The common practical algorithm relies on a union-find representation of finite sets to record equivalent symbolic types. Every set contains at least one symbolic type that is just a variable. This algorithm runs in essentially linear time.
- A linear algorithm exists but it is not as efficient for problems of practical size as the union-find based algorithm.

- Question: is the reconstructed type unique? Many □-calculus programs have multiple typings. Consider the program (□*x x*). It can be assigned the type □□ □, for any type □ □ *T* , *e.g.*, (int □ int), (int □ int) □ (int □ int), …

  The type reconstruction algorithm deftly addresses this problem by returning the most general symbolic typing; all of the possible *ground* typings (containing no type variables) are substitution instances of this typing. For this reason, the typing produced by the type reconstruction algorithm is called the *principal typing* (or type) of the program.

Type Systems

Robin Milner's Creative Leaps

- **The simple type system for the λ-calculus is truly onerous because *polymorphic* functions (those with variables in their principal types) have to be rewritten for each different typing. The original Pascal language suffers from precisely this problem.**

Milner recognized that a surprisingly useful form of polymorphism could be added to the (simply) typed λ-calculus by adding a let construct to the language family. The extension adds one new form to the family syntax

***M* ::= … | (let (*x* *M*) *M*)**

Given an expression of the form

(let (*x* *M*) *N*)

***x* can be used polymorphically in *N* *without breaking the principal typing property.* Type reconstruction can first infer the principal type of *M* and subsequently use a renamed version of this type (a fresh name for each distinct type variable) for each distinct occurrence of *x* in *N*.**

Type Systems

Robin Milner's Creative Leaps continued

Explanation: in essence, the definition of *x* is treated like a macro (abbreviation) that is replicated for each occurrence of *x* in *N.* Of course, a language implementation only needs one copy of the code for *M* provided that all the different instantiations of the principal type use the same data layout.

Example:

In our simple □ language: (let (*i* (□*x x*)) (((*i* +) (*i* 2)) 2))

In a richer language of the family:

(let (append (□ *x*,*y* (if (empty? *x*) y (cons (first *x*) (append (rest *x*) *y*))))

(append (cons (append (cons 1 empty) (cons 2 empty)) empty) empty))

Note: empty and cons are polymorphic constants; let is recursive.

_ Milner also realized he could rigorously prove a that typable programs cannot generate certain errors. Type reconstruction proves that run-time type-errors *cannot* occur --- provided that we define the notion of type-error rather narrowly. A similar theorem holds for Java (we think).