Simple Types

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.

With content from Arvind and Adam Chlipala. Used with permission.

September 23, 2015

Before we Start
Some more Coq

Induction over natural numbers

N ::= O | S N

Induction principle:
To prove $\forall n \in N.\ P(n)$:

Base case:
Show P(0).

Inductive case:
Assume P(*n*).
Show P(S(*n*)).

Structural Induction

T ::= Leaf | Node T T

Induction principle:
To prove ∀ *t* ∈ T. P(*t*):

Base case:
Show P(Leaf).

Inductive case:
Assume P(*t1*).
Assume P(*t2*).
Show P(Node *t1 t2*).

Another Example

E ::= Const N | Plus E E | Times E E

Induction principle:
To prove ∀ *e* ∈ E. P(*e*):

Base case:
Show P(Const n).

Inductive case 1:
Assume P(e*1*).
Assume P(e*2*).
Show P(Plus e*1* e*2*).

Inductive case 2:
Assume P(e*1*).
Assume P(e*2*).
Show P(Times e*1* e*2*).

Proofs as a Datatype

$$\frac{}{\textbf{even}(0)} \qquad\qquad \frac{\textbf{even}(n)}{\textbf{even}(n+2)}$$

Example Derivations:

$$\frac{}{\textbf{even}(0)} \qquad \frac{\textbf{even}(0)}{\textbf{even}(2)} \qquad \frac{\frac{\textbf{even}(0)}{\textbf{even}(2)}}{\textbf{even}(4)}$$

...and so on for all even numbers.

even ::= Even0 : **even**(0)
| Even2 (**even** *n*) : **even**(*n*+2)

Examples:
EvenO : even(0)
Even2(EvenO) : even(2)
Even2(Even2(EvenO)) : even(4)

Induction on Proofs (***Rule Induction***)

$$\frac{}{\textbf{even}(0)} \qquad \frac{\textbf{even}(n)}{\textbf{even}(n+2)}$$

even ::= Even0 : **even**(0)
| Even2 (**even** *n*) : **even**(*n*+2)

Induction principle:
To prove $\forall\, n \in N.\ \text{even}(n) \Rightarrow P(n)$:

Because I have a rule that if (n) is even, it lets me prove that (n+2) is even

Base case:
Show P(0).

Inductive case:
Assume P(*n*).
Show P(*n*+2).

Because I have a rule that lets me prove even(0) so I need to show that P(0) holds.

Also called Induction on the Structure of Derivations

More Rule Induction

$$\frac{}{\textbf{eval}(\text{Const } n, n)} \qquad \frac{\textbf{eval}(e1, n1) \qquad \textbf{eval}(e2, n2)}{\textbf{eval}(\text{Plus e1 e2, n1 + n2})}$$

eval ::= EvConst : **eval** (Const *n*, *n*)
| EvPlus (**eval**(*e1*, *n1*)) (**eval**(*e2*, *n2*))
: **eval** (Plus *e1 e2*, *n1* + *n2*)

Induction principle:
To prove ∀ *e* ∈ E, *n* ∈ N. eval *e n* ⇒ P(*e*, *n*):

Base case:
Show P(Const *n*, *n*).

Inductive case:
Assume P(*e1*, *n1*).
Assume P(*e2*, *n2*).
Show P(Plus *e1 e2*, *n1* + *n2*).

More Tactics

- induction N:
  - Induction on the derivation of the [N]th hypothesis in the conclusion
  - (numbering goes left to right and starts at 1).
- destruct E
  - Do case analysis on the constructor used to build term [E].
- assumption
  - Prove a conclusion that matches a known hypothesis; like doing apply H where H is the known hypothesis.
- eapply thm
  - Like apply, but leaves placeholders for theorem parameters that are not known yet.
- eassumption
  - Like assumption, but also learns values for placeholders in the process.
- rewrite <- H
  - Like [rewrite], but rewrites right-to-left.

More powerful tactics

- generalize thm1,...,thmN
  - Bring the statements of a set of theorems into the goal explicitly so that other tactics don’t need to deduce them manually.
- firstorder
  - Magic heuristic procedure for proofs based on first-order logic rules.
  - (It’s undecidable in general, so don’t get too excited.)

And now some types!

Why Types

```
let
    f x = if x then 5 else 2
in
    f 5+1
```

```
let
    f x = if x then 5 else 2
in
    f 6
```

```
let
    f x = if x then 5 else 2
in
    if 6 then 5 else 2
```

!!

What to do in this situation?

- Options
  1) Leave it up to the implementation
     - that's the C approach
     - is it a good idea?
  2) Provide a mechanism to identify and rule out such "bad" programs
     - programs can only run if you can prove they will execute to completion according to the semantics of the language
     - type systems will allow us to do this!
  3) Prescribe correct behavior for every program
     - untyped λ-calculus works like this
     - do any practical languages do this?
     - type systems are useful in this situation too.

Self-application and Paradoxes

Self application, i.e., (x x) is dangerous.

Suppose:

$u \equiv \lambda y.$ if (y y) = a then b else a

What is (u u) ?

(u u) $\rightarrow$ if (u u) = a then b else a

Contradiction!!!

This was one of the original motivations for types

What is a type system

- Narrow View
  - It's a mechanism for ensuring that variables only take values from predefined sets
    - Ex. Integers, Strings, Characters
  - A mechanism for avoiding unchecked errors
    - by ruling out programs with undefined behaviors
    - by specifying how a program should fail (eg. NullPointerException)
- Expansive View
  - It's a light-weight proof system and annotation mechanism for efficiently checking for a specific property of interest
  - Address bugs that go beyond corner-cases in the semantics
    - Information flow violations
    - deadlocks
    - etc, etc, etc

What are Types?

- A method of classifying objects (values) in a language

  $x :: \tau$

  says object x has type τ or object x belongs to a type τ

- τ denotes a set of values.

  This notion of types is different from types in languages like C, where a type is a storage class specifier.

Type Correctness

- If x :: τ then only those operations that are appropriate to set τ may be performed on x.

- A program is type correct if it never performs a wrong operation on an object.

  - Add an Int and a Bool
  - Head of an Int
  - Square root of a list

Type Safety

- A language is type safe if only type correct programs can be written in that language.

- Most languages are not type safe, i.e., have “holes” in their type systems.

  Fortran: Equivalence, Parameter passing
  Pascal: Variant records, files
  C, C++: Pointers, type casting

  However, Java, Ada, CLU, ML, Id, Haskell, Bluespec, etc. are type safe.

Type Declaration vs Reconstruction

- Languages where the user must declare the types
  - CLU, Pascal, Ada, C, C++, Fortran, Java

- Languages where type declarations are not needed and the types are reconstructed at run time
  - Scheme, Lisp

- Languages where type declarations are generally not needed but allowed, and types are reconstructed at compile time
  - ML, Id, Haskell, pH, Bluespec

A language is said to be statically typed if type-checking is done at compile time

Polymorphism

- In a monomorphic language like Pascal, one defines a different length function for each type of list
- In a polymorphic language like ML, one defines a polymorphic type (list t), where t is a type variable, and a single function for computing the length
- Haskell and most modern functional languages have polymorphic types and follow the Hindley-Milner type system.

Simple types = Non polymorphic types

more on polymorphic types – next time ...

Formalizing a Type System

Formalizing a type system

- The type system is almost never orthogonal to the semantics of the language
  - The types in a program can affect its behavior (e.g. operator overloading)

- We don't define the type system in isolation, we define a typed language including definitions of
  - The syntax
  - dynamic semantics (e.g. operational semantics)
  - static semantics
    - also known as typing rules
    - describe how types are assigned to elements in a program
  - type soundness argument
    - describe the relationship between static and dynamic semantics

Basic notation

- The type system assigns types to elements in the language
  - basic notation: e : T (e is of type T)
  - What is the type of :

    5

    ?

- The types of some elements depends on the environment
  - basic notation $\Gamma \vdash e : T$

    (Given environment Γ , we can derive that e is of type T)
  - An environment associates types with free variables
  - This is called a Judgment
  - Ex.

    $$x\!:\!int\,,y\!:\!int\ \vdash\ x+y : int$$

Static Semantics

- Typing rules
  - Typing rules tell us how to derive typing judgments
  - Very similar to derivation rules in Big Step OS

$$\frac{premises}{Judgment}$$

- Ex. Language of Expressions

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash N : int} \qquad \frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 + e2 : int}$$

Ex. Language of Expressions

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash N : int} \qquad \frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 + e2 : int}$$

- Show that the following Judgment is valid

$$x{:}\, int, y{:}\, int \vdash x + (y + 5) : int$$

$$\frac{x{:}\, int, y{:}\, int \vdash x{:}\, int \qquad x{:}\, int, y{:}\, int \vdash (y + 5) : int}{x{:}\, int, y{:}\, int \vdash x + (y + 5) : int}$$

$$\frac{\dfrac{x{:}\, int \in x{:}\, int, y{:}\, int}{x{:}\, int, y{:}\, int \vdash x{:}\, int} \qquad \dfrac{x{:}\, int, y{:}\, int \vdash y{:}\, int \qquad x{:}\, int, y{:}\, int \vdash 5 : int}{x{:}\, int, y{:}\, int \vdash (y + 5) : int}}{x{:}\, int, y{:}\, int \vdash x + (y + 5) : int}$$

Simply Typed λ Calculus (F_1)

- Basic Typing Rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1\ e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

- Extensions

$$\frac{}{\Gamma \vdash N : int} \qquad \frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 + e2 : int} \qquad \frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 = e2 : bool}$$

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash if\ e\ then\ e_t\ else\ e_f : \tau}$$

Example

- Is this a valid typing judgment?

$$\vdash (\lambda x\colon bool\ \lambda y\colon int\ \ if\ x\ then\ y\ else\ y+1)\colon bool \to int \to int$$

- How about this one?

$$\vdash (\lambda x\colon int\ \ \lambda y\colon bool\ x + y)\colon int \to bool \to int$$

Example

- What's the type of this function?

  (λ f. λ x. if x = 1 then x else (f f (x-1)) * x)

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash (\lambda x:\tau_1\ e):\tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1:\tau' \rightarrow \tau \quad \Gamma \vdash e_2:\tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

$$\frac{}{\Gamma \vdash N : int} \qquad \frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 + e2 : int} \qquad \frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 = e2 : bool}$$

$$\frac{\Gamma \vdash e: bool \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash if\ e\ then\ e_t\ else\ e_f : \tau}$$

  – Hint: This IS a trick question

Simply Typed λ Calculus (F_1)

- We have defined a really strong type system on λ-calculus
  - It's so strong, it won't even let us write non-terminating computation
  - We can actually prove this!

6.820 Fundamentals of Program Analysis
Fall 2015