Lecture 18: Axiomatic Semantics & Type Safety

CSCI 131
Fall, 2011

Kim Bruce

Axiomatic Rules

- Assignment axiom:
  - {P [expression / id]} `id := expression` {P}
  - Ex: {a+47 > 0} `x := a+47` {x > 0}
  - {x > 1} `x := x - 1` {x > 0}
- While rule:
  - If {P & B} `stats` {P}, then
    {P} `while B do stats` {P & not B}
  - P is *invariant* of loop.

Axiomatic Rules

- Composition rule:
  - If {P} `S1` {Q}, {R} `S2` {T}, and Q ⇒ R, then {P} `S1; S2` {T}
- Conditional rule:
  - If {P & B} `S1` {Q}, {P & not B} `S2` {Q},
    then {P} `if B then S1 else S2` {Q}
- Consequence rule:
  - If P ⇒ Q, R ⇒ T, and {Q} S {R}, then {P} S {T}

Correctness using Axioms & Rules

- Due to Bob Floyd & Tony Hoare
- Prove {precondition} `Prog` {postcondition}
- Usually work backwards from postcondition.

```
{Pre: exponent₀ >= 0}
base <- base₀
exponent <- exponent₀
ans <- 1
while exponent > 0 do
{assert:  ans * (base ** exponent) = base₀ ** exponent₀}
{           & exponent >= 0}
    if odd(exponent) then
            ans<- ans*base
            exponent <- exponent - 1
    else
            base <- base * base
            exponent <- exponent div 2
    end if
end while
{Post: exponent = 0 & ans = base₀ ** exponent₀}
```

Steps in Proof

- Show
  `ans * (base ** exponent) = base₀ ** exponent₀`
  `& exponent >= 0`
  is loop invariant
- Show postcondition follows from
  `(ans * (base ** exponent) = base₀ ** exponent₀`
  `& exponent >= 0) & not(exponent > 0)`
- Push invariant back to beginning of program.

Type Safety

- Is there any connection between type checking rules and semantics?
- If E ⊢ e: T, what does that say about computation (e, *env*) ⇒ τv?
- If E and *env* "correspond", then expect τv: T

Typed PCF

- T :: = Int | Bool | T -> T
- Provide identifiers w/type when introduced.
- e ::= x | n | true | false | succ | pred | iszero | if e then e else e | (fn (x:T) => e) | (e e) | rec (x:T) => e *Ignore recursion here!*

Type-checking Rules

E is *type environment*: identifiers → types

E ⊢ n: Int, if n is an integer

E ⊢ true: Bool, E ⊢ false: Bool

E ⊢ succ: Int → Int, E ⊢ pred: Int → Int

E ⊢ iszero: Int → Bool

E ⊢ x: E(x)

More Type-Checking Rules

$$\frac{E \vdash e: \text{Bool},\ E \vdash e_1: T,\ E \vdash e_2: T}{E \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T}$$

$$\frac{E \vdash f: T \to U,\ \ E \vdash x: T}{E \vdash (f\ x): U}$$

$$\frac{E[x:T] \vdash \text{body}: U}{E \vdash (\text{fn } (x:T) \Rightarrow \text{body}): T \to U}$$

Computation Rules

(id, *env*) ⇒ *env*(id) (n, *env*) ⇒ $n\smile$, for n an int

(true, *env*) ⇒ *true*$\smile$ (false, *env*) ⇒ *false*$\smile$

(succ, *env*) ⇒ *succ* (pred, *env*) ⇒ *pred*

(iszero, *env*) ⇒ *iszero*

$$\frac{(b, env) \Rightarrow true\smile,\ \ (e_1, env) \Rightarrow \tau v}{(\text{if } b \text{ then } e_1 \text{ else } e_2, env) \Rightarrow \tau v}$$ ....

Computation Rules

((fn (x:T) => e), *env*) ⇒< fn (x:T) => e, *env* >

$$\frac{(e_1, env) \Rightarrow < \text{fn } (x:T) \Rightarrow e_3, env' >\ \ (e_2, env) \Rightarrow \tau v_2,\ \ (e_3, env'\ [\tau v_2/x]) \Rightarrow \tau v}{((e_1\ e_2), env) \Rightarrow \tau v}$$

Typing Values

$\vdash_v \underline{n}$: Int, for n an integer

$\vdash_v \underline{true}$: Bool

$\vdash_v \underline{false}$: Bool

$\vdash_v \underline{succ}$: Int → Int

$\vdash_v \underline{pred}$: Int → Int

$\vdash_v \underline{iszero}$: Int → Bool

Environment Compatibility

$$\frac{E' \vdash \text{fn } (x{:}T) \Rightarrow e{:}\ T \to U}{\vdash_v \langle \text{fn } (x{:}T) \Rightarrow e, env \rangle{:}\ T \to U}$$

for E' s.t E' and *env* are $\underline{compatible}$

E and *env* are $\underline{compatible}$ iff

domain(E) = domain(*env*), and

for all x in domain(*env*), $\vdash_v env(x)$: E(x)

Safety

Theorem: (Subject Reduction) Let E and *env* be compatible environments.

Let e be a term of typed PCF. If E ⊢ e: T and (e,*env*) ⇒ v, then $\vdash_v v$: T.

Proof: By induction on proof of E ⊢ e: T

Proof

Conditional: S'pose E ⊢ if b then e1 else e2: T because
E ⊢ b: boolean, E⊢e1: T, and E⊢e2: T.
Two cases depending on the evaluation of b.

Case 1: (b, *env*) ⇒ $\underline{true}$.

Then if (e1,*env*) ⇒ v, it follows that
(if b then e1 else e2, *env*) ⇒ v.

By induction and E⊢e1: T, it follows that $\vdash_v v$: T, which is all we need.

Case 2: (b, *env*) ⇒ $\underline{false}$ is similar.

Application case/*succ*:

Let e be (e1 e2) and E ⊢ (e1 e2): U because
E ⊢ e1: T → U and E ⊢ e2: T.
Suppose also that (e1, *env*) ⇒ *f* and (e2, *env*) ⇒ $v2$.

By induction, $\vdash_v f$: T → U and $\vdash_v v2$: T.
Need to show final result has type U.

Subcase: *f* is *succ*. Then T = U = Int.
Because $\vdash_v v2$: Int, $v2$ must be an integer.
By the semantics rule, the result is $v2+1$, which has type Int, as desired, as E ⊢ (e1 e2): Int.

Closure case

Subcase: *f* is < fn (x:T) => e3, *env*' >
By induction, $\vdash_v$ < fn (x:T) => e3, *env*' >: T → U &
∃ E' s.t. E', *env*' compatible & E'⊢ fn (x:T) => e3: T → U

Spose (e3, *env*' [$v2$/x]) ⇒ v. Most show $\vdash_v v$: U.
By above, must have had E'[x:T] ⊢ e3: U.

E' & *env*' compatible ⇒ E'[x:T] & *env*' [$v2$/x] compatible
So from (e3, *env*' [$v2$/x]) ⇒ v,
it follows that $\vdash_v v$: U.

Type Safety

- Errors have been made in type systems.
  - See examples in OO languages
- Need to verify that type system is consistent with semantics.
- Progress Lemma not shown here, but also important

Scala: First steps

Scala Design Goals

- Design started in 2003 by Martin Odersky
- Better language support for component software.
- Scalability
- Mechanisms for abstraction, composition, and decomposition.

Innovations

- Path dependent types give strength of dependent types (usually undecidable!)
- Modular mixin composition (supports traits)
- Views (implicit conversions)

Features

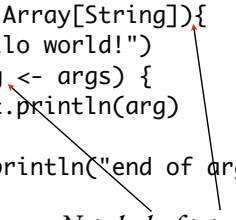
- Every value is an object.
- Blend object-oriented and functional programming
- Seamless interoperability with Java (and .net?)
- Powerful abstraction concepts for types & values
- Decomposition of objects by pattern matching

Concrete Features

- Object (singleton) as well as class definitions
- Declarations as x: T, use Unit instead of void
- All defs start with key word
- No need for semicolons or "return"
- Local type inference allows omitting many types, including with generics.
- For comprehensions rather than traditional for

Running Scala

- Scala installed on xserv.cs.pomona.edu and all lab Macs. Should be at HMC, too, but check.
- Read tutorial to see how to run programs:
  - http://www.scala-lang.org/docu/files/ScalaTutorial.pdf
- Running within eclipse slightly buggy
  - See link from course web page and follow directions carefully!
  - Probably will need to increase eclipse memory size in eclipse.ini -- see Troubleshooting page
  - On Mac, eclipse.ini inside Eclipse.app folder

Hello World!

```
package helloWorld

object HelloWorld {

  def main(args:Array[String]){
    println("Hello world!")
    for (val arg <- args) {
      System.out.println(arg)
    }
    System.out.println("end of args");
  }
}
```