Type Checking, Inference, & Elaboration

CS4410: Spring 2013

Statics

- After parsing, we have an AST.
- Critical issue:
  - not all operations are defined on all values.
  - e.g., (3 / 0), sub("foo",5), 42(x)
- Options
  1. don't worry about it (C, C++, etc.)
  2. report errors at run time (Scheme)
  3. rule out ill-formed expressions at compile time (ML)

Type Soundness

- Construct a model of the source language
  - i.e., interpreter
  - This tells us where operations are partial.
  - And partiality is different for different languages (e.g., "foo" + "bar" may be meaningful in some languages, but not others.)
- Construct a function TC: AST -> bool
  - when true, should ensure interpreting the AST does not result in an undefined operation.
- Prove that TC is correct.

Simple Language:

```
type tipe =
  Int_t
| Fn_t of tipe*tipe
| Pair_t of tipe*tipe

type exp =
  Var of var | Int of int
| Plus_i of exp*exp
| Lambda of var * tipe * exp
| App of exp*exp
| Pair of exp * exp
| Fst of exp | Snd of exp
```

Interpreter:

```
let rec interp (env:var->value)(e:exp) =
  match e with
   | Var x -> env x
   | Int I -> Int_v i
   | Plus_i(e1,e2) ->
       (match interp env e1, interp env e2 of
         | Int i, Int j -> Int_v(i+j)
         | _,_ => error())
   | Lambda(x,t,e) => Closure_v{env=env,code=(x,e)}
   | App(e1,e2) =>
      (match interp env e1, interp env e2 of
         | Closure_v{env=cenv,code=(x,e)},v ->
               interp (extend cenv x v) e
         | _,_ -> error()
```

Type Checker:

```
let rec tc (env:var->tipe) (e:exp) =
  match e with
   | Var x -> env x
   | Int _ -> Int_t
   | Plus_i(e1,e2) ->
       (match tc env e1, tc env e with
         |  Int_t, Int_t -> Int_t
         | _,_ => error())
   | Lambda(x,t,e) ->
       Fn_t(t,tc (extend env x t) e)
   | App(e1,e2) ->
      (match (tc env e1, tc env e2) with
         | Fn_t(t1,t2), t ->
             if (t1 != t) then error() else t2
         | _,_ -> error())
```

Notes:

- In the interpreter, we only evaluate the body of a function when it's applied.

- In the type-checker, we always check the body of the function (even if it's never applied.)

- Because of this, we must *assume* the input has some type (say t_1) and reflect this in the type of the function (t_1 -> t_2).

- Dually, at a call site (e_1 e_2), we don't know what *closure* we're going to get.

- But we can calculate e_1's type, check that e_2 is an argument of the right type, and also determine what type e_1 will return.

Growing the language

```
type tipe = ... | Bool_t

type exp = ... |
  True | False | If of exp*exp*exp

let rec interp env e = ...
| True -> True_v
| False -> False_v
| If(e1,e2,e3) ->
   (match interp env e1 with
     True_v -> interp env e2
   | False_v -> interp env e3
   | _ => error())
```

Type-Checking

```
let rec tc (env:var->tipe) (e:exp) =
  match e with
   ...
   | True -> Bool_t
   | False -> Bool_t
   | If(e1,e2,e3) ->
    (let (t1,t2,t3) = (tc env e1,tc env e2,tc env e3)
     in
        match t1 with
        | Bool_t ->
            if (t2 != t3) then error() else t2
        | _ => error())
```

Refining Types

We can easily add new types that distinguish different subsets of values.

```
type tipe =
   ...
| True_t | False_t | Bool_t
| Pos_t | Neg_t | Zero_t | Int_t
| Any_t
```

Modified Type-Checker

```
let rec tc (env:var->tipe) (e:exp) =
    ...
   | True -> True_t
   | False -> False_t
   | If(e1,e2,e3) ->
       (match tc env e1 with
          True_t -> tc env e2
         | False_t -> tc env e3
         | Bool_t ->
            (let (t2,t3) = (tc env e2, tc env e3)
             in
                lub t2 t3)
         | _ => error())
```

Least Upper Bound

```
let lub t1 t2 =
  match t1, t2 with
   | True_t  (Bool_t|False_t) -> Bool_t
   | False_t,(Bool_t|True_t) -> Bool_t
   | Zero_t, (Neg_t|Pos_t|Int_t) -> Int_t
   | Neg_t,  (Zero_t|Pos_t|Int_t) -> Int_t
   | Pos_t,  (Zero_t|Neg_t|Int_t) -> Int_t
   | _,_ -> if (t1 = t2) then t1 else Any_t
```

Refining Integers into Zero,Neg,Pos

```
let rec tc (env:var->tipe) (e:exp) =
    ...
   | Int 0 -> Zero_t
   | Int i -> if i < 0 then Neg_t else Pos_t
   | Plus(e1,e2) ->
    (match tc env e1, tc env e2 with
     | Zero_t,t2 -> t2
     | t1,Zero_t -> t1
     | Pos_t,Pos_t -> Pos_t
     | Neg_t,Neg_t -> Neg_t
     | (Neg_t|Pos_t|Int_t),Int_t -> Int_t
     | Int_t,(Neg_t|Pos_t) -> Int_t
     | _,_ -> error())
```

Subtyping as Subsets

- If we think of types as *sets* of values, then a subtype corresponds to a subset and lub corresponds to union.
- e.g., Pos_t <= Int_t since every positive integer is also an integer.
- For conditionals, we want to find the *least* type of the types the two branches might return.
  - Adding "Any_t" ensures there is a type.
  - (Not always a good thing to have…)
  - Need NonPos_t, NonNeg_t and NonZero_t to get *least* upper bounds.

Extending Subtyping:

- What about pairs?
  - $(T_1 * T_2) <= (U_1 * U_2)$ when $T_1 <= U_1$ and $T_2 <= U_2$
  - But only when immutable!
  - Why?
- What about functions?
  - $(T_1 \text{ -> } T_2) <= (U_1 \text{ -> } U_2)$ when $U_1 <= T_1$ and $T_2 <= U_2$.
  - Why?

Problems with Mutability:

```
let f(p :ref(Pos_t)) =
  let q :ref(Int_t) = p
  in
    q := 0;
    42 div (!p)
```

Another Way to See it:

- Any shared, mutable data structure can be thought of as an immutable record of pairs of methods (with hidden state):
  - `p : ref(Pos_t) =>`
  - ```
    p : { get: unit -> Pos_t,
          set: Pos_t -> unit }
    ```
- When is ref(T) <= ref(U)? When:
  - `unit->T <= unit->U` or `T <= U` and
  - `T->unit <= U->unit` or `U <= T`.
  - Thus, only when `T = U`!
    ```

N-Tuples and Simple Records:

- $(T_1 * \ldots * T_n * T_{n+1}) <= (U_1 * \ldots * U_n)$ when $T_i <= U_i$.
  - Why?

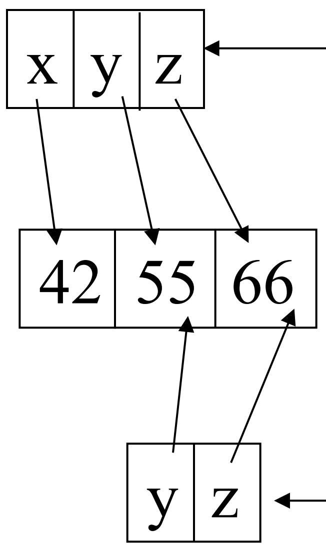
- Non-Permutable Records: $\{\ell_1:T_1,\ldots,\ell_n:T_n,\ell_{n+1}:T_{n+1}\} <= \{\ell_1:U_1,\ldots, \ell_n:U_n\}$ when $T_i <= U_i$.
  - Assumes {x:int,y:int} != {y:int,x:int}
  - That is, the position of a label is independent of the rest of the labels in the type.
  - In SML (or for Java interfaces) this is *not* the case.

SML-Style Records:

- Compiler sorts by label.
- So if you write {y:int,z:int,x:int}, the compiler immediately rewrites it to {x:int,y:int,z:int}.
- So you need to know all of the labels to determine their positions.
- Consider: {y:int,z:int,x:int} <= {y:int,z:int} but {y,z,x} == {x,y,z} ~~<=~~ {y,z}

If you want both:

- If you want permutability & dropping, you need to either copy or use a dictionary:



p = {x=42,y=55,z=66}:{x:int,y:int,z:int}

q : {y:int,z:int} = p

Type Inference

```
let rec tc (env:(var*tipe) list) (e:exp) =
  match e with
  | Var x -> lookup env x
  | Lambda(x,e) ->
      (let t = guess()
       in
          Fn_t(t,tc (extend env x t) e))
  | App(e1,e2) ->
     (match tc env e1, tc env e2 with
       | Fn_t(t1,t2), t ->
           if t1 != t then error() else t2
       | _,_ => error())
```

Extend Types with Guesses:

```
type tipe =
   Int_t
| Fn_t of tipe*tipe
| Guess of (tipe option ref)

fun guess() = Guess(ref None)
```

Must Handle Guesses

```
| Lambda(x,e) ->
    let t = guess()
    in
       Fn_t(t,tc (extend env x t) e)
| App(e1,e2) ->
  (match tc env e1, tc env e2 with
   | Fn_t(t1,t2), t ->
         if t1 != t then error() else t2
   | Guess (r as ref None), t ->
         let t2 = guess() in
           r := Some(Fn_t(t,t2)); t2
   | Guess (ref Some (Fn_t(t1,t2))), t ->
         if t1 != t then error() else t2
```

Cleaner:

```
let rec tc (env: (var*tipe) list) (e:exp) =
  match e with
   | Var x -> lookup env x
   | Lambda(x,e) ->
       let t = guess()
       in
           Fn_t(t,tc (extend env x t) e)
   | App(e1,e2) ->
       let (t1,t2) = (tc env e1, tc env e2) in
       let t = guess()
       in
          if unify t1 (Fn_t(t2,t)) then t
          else error()
```

Where:

```
let rec unify (t1:tipe) (t2:tipe):bool =
  if (t1 = t2) then true else
  match t1,t2 with
  | Guess(ref(Some t1')), _ -> unify t1′ t2
  | Guess(r as (ref None)), t2 ->
         (r := t2; true)
  | _, Guess(_) -> unify t2 t1
  | Int_t, Int_t -> true
  | Fn_t(t1a,t1b), Fn_t(t2a,t2b)) ->
     unify t1a t2a && unify t1b t2b
```

Subtlety

- Consider: **`fun x => x x`**
- We guess **`g1`** for **`x`**
  - We see **`App(x,x)`**
  - recursive calls say we have **`t1=g1`** and **`t2=g1`**.
  - We guess **`g2`** for the result.
  - And **`unify(g1,Fn_t(g1,g2))`**
  - So we set **`g1 := Some(Fn_t(g1,g2))`**
  - What happens if we print the type?

Fixes:

- Do an "occurs" check in unify:

```
let rec unify (t1:tipe) )t2:tipe):bool =
  if (t1 = t2) then true else
  match (t1,t2) with
    (Guess(r as ref None),_) =>
       if occurs r t2 then error()
       else (r := Some t2; true)
  | …
```

- Alternatively, be careful not to loop anywhere.
  - In particular, when comparing t1 = t2, we must code up a *graph* equality, not a tree equality.

Polymorphism:

- Consider: **`fun x => x`**
- We guess **`g1`** for **`x`**
  - We see **`x`**.
  - So **`g1`** is the result.
  - We return **`Fn_t(g1,g1)`**
  - **`g1`** is unconstrained.
  - We could constraint it to **`Int_t`** or **`Fn_t(Int_t,Int_t)`** or *any* type.
  - In fact, we could re-use this code at any type.

ML Expressions:

```
type exp =
  Var of var
| Int of int
| Lambda of var * exp
| App of exp*exp
| Let of var * exp * exp
```

Naïve ML Type Inference:

```
let rec tc (env: (var*tipe) list) (e:exp) =
  match e with
  | Var x -> lookup env x
  | Lambda(x,e) ->
      let t = guess() in
        Fn_t(t,tc (extend env x t) e) end
  | App(e1,e2) ->
      let (t1,t2) = (tc env e1, tc env e2) in
      let t = guess()
      in if unify t1 (Fn_t(t2,t)) then t
         else error()
  | Let(x,e1,e2) =>
      (tc env e1; tc env (substitute(e1,x,e2))
```

Example:

```
let id = fn x => x
in
  (id 3, id "fred")
end

====>

((fun x => x) 3, (fun x => x) "fred")
```

Better Approach (DM):

```
type tvar = string

type tipe =
  Int_t
| Fn_t of tipe*tipe
| Guess of (tipe option ref)
| Var_t of tvar

type tipe_scheme =
  Forall of (tvar list * tipe)
```

ML Type Inference

```
let rec tc (env:(var*tipe_scheme) list) (e:exp) =
  match e with
   | Var x -> instantiate(lookup env x)
   | Int _ -> Int_t
   | Lambda(x,e) ->
       let t = guess() in
       Fn_t(t,tc (extend env x (Forall([],t)) e)
   | App(e1,e2) ->
       let (t1,t2,t) = (tc env e1,tc env e2,guess())
       in if unify(t1,Fn_t(t2,t)) then t else error()
   | Let(x,e1,e2) ->
       let s = generalize(env,tc env e1) in
       tc (extend env x s) e2 end
```

Instantiation

```
let instantiate(s:tipe_scheme):tipe =
  let val Forall(vs,t) = s
      val vs_and_ts : (var*tipe) list =
          map (fn a => (a,guess())) vs
  in
      substitute(vs_and_ts,t)
  end
```

Generalization:

```
let generalize(e:env,t:tipe):tipe_scheme =
  let t_gs = guesses_of_tipe t in
  let env_list_gs =
       map (fun (x,s) -> guesses_of s) e in
  let env_gs = foldl union empty env_list_gs
  let diff = minus t_gs env_gs in
  let gs_vs =
       map (fun g -> (g,freshvar())) diff in
  let tc = subst_guess(gs_vs,t)
  in
      Forall(map snd gs_vs, tc)
   end
```

Explanation:

- Each let-bound value is generalized.
  - e.g., g->g generalizes to Forall a.a -> a.
- Each use of a let-bound variable is instantiated with fresh guesses:
  - e.g., if f:Forall a.a->a, then in f e, then the type we assign to f is g->g for some fresh guess g.
- But we can't generalize guesses that might later become constrained.
  - Sufficient to filter out guesses that occur elsewhere in the environment.
  - e.g., if the expression has type g1->g2 and y:g1, then we might later use y in a context that demands it has type int, such as y+1.

Effects:

- The algorithm given is equivalent to substituting the let-bound expression.
- But in ML, we evaluate CBV, not CBN!

```
let id = (print "Hello"; fn x => x)
in
  (id 42, id "fred”)

!=

((print "Hello";fn x=>x) 42,
  (print "Hello";fn x=>x) "fred")
```

Problem:

```
let r = ref (fn x=>x)
      (* r : Forall 'a.ref('a->'a)   *)
in
  r := (fn x => x+1); (* r:ref(int->int) *)
  (!r)("fred") (* r:ref(string->string) *)
```

"Value Restriction"

- When is **`let x=e1 in e2`** equivalent to **`subst(e1,x,e2)`**?
- If e1 has no side effects.
  - reads/writes/allocation of refs/arrays.
  - input, output.
  - non-termination.
- So only generalize when e1 is a *value*.
  - or something easy to prove equivalent to a value.

Real Algorithm:

```
let rec tc (env:var->tipe_scheme) (e:exp) =
  match e with
  ...
| Let(x,e1,e2) ->
    let s =
         if may_have_effects e1 then
            Forall([],tc env e1)
         else generalize(env,tc env e1)
    in
      tc (extend env x s) e2
    end
```

Checking Effects:

```
let rec may_have_effects e =
  match e with
  | Int _ -> false
  | Var _ -> false
  | Lambda _ -> false
  | Pair(e1,e2) ->
    may_have_effects e1||may_have_effects e2
  | App _ -> true
```