

Type Checking, Inference, & Elaboration

CS153: Compilers

Greg Morrisett

Statics

- After parsing, we have an AST.
- Critical issue:
 - not all operations are defined on all values.
 - e.g., (3 div 0), sub("foo",5), 42(x)
- Options
 1. don't worry about it (C, C++, etc.)
 2. force all operations to be total (Scheme)
 - e.g., (3 div 0) -> 3 , 42(x) -> 124
 1. try to rule out ill-formed expressions (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 \rightarrow bool
 - when true, should ensure interpreting the AST does not result in an undefined operation.
- Prove that TC is correct.

Simple Language:

```
datatype tipe =
```

```
  Int_t
```

```
| Fn_t of tipe*tipe
```

```
| Pair_t of tipe*tipe
```

```
datatype 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:

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

Type Checker:

```
fun tc (env:var->type) (e:exp) =
  case e of
    Var x => env x
  | Int _ => Int_t
  | Plus_i(e1,e2) =>
    (case (tc env e1, tc env e2) of
      (Int_t, Int_t) => Int_t
      | (_,_) => error())
  | Lambda(x,t,e) =>
    Fn_t(t,tc (extend env x t) e)
  | App(e1,e2) =>
    (case (tc env e1, tc env e2) of
      (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 ($t_1 \rightarrow t_2$).
- 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

```
datatype tipe = ... | Bool_t
```

```
datatype exp = ... |  
  True | False | If of exp*exp*exp
```

```
fun interp env e = ...  
| True => True_v  
| False => False_v  
| If(e1,e2,e3) =>  
  (case (interp env e1) of  
    True_v => interp env e2  
  | False_v => interp env e3  
  | _ => error())
```

Type-Checking

```
fun tc (env:var->type) (e:exp) =  
  case e of  
    ...  
  | True => Bool_t  
  | False => Bool_t  
  | If(e1,e2,e3) =>  
    let val (t1,t2,t3) =  
      (tc env e1,tc env e2,tc env e3)  
    in  
      case t1 of  
        Bool_t =>  
          if (t2 != t3) then error() else t2  
        | _ => error()  
    end  
end
```

Refining Types

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

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

Modified Type-Checker

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

Least Upper Bound

```
fun lub (True_t, (Bool_t|False_t)) = Bool_t
  | lub (False_t, (Bool_t|True_t)) = Bool_t
  | lub (Zero_t, (Neg_t|Pos_t|Int_t)) = Int_t
  | lub (Neg_t, (Zero_t|Pos_t|Int_t)) = Int_t
  | lub (Pos_t, (Zero_t|Pos_t|Int_t)) = Int_t
  | lub (t1, t2) = if (t1 = t2) then t1 else Any_t
```

Integers:

```
fun tc (env:var->tipe) (e:exp) =
  ...
  | Int 0 => Zero_t
  | Int i => if i < 0 then Neg_t else Pos_t
  | Plus(e1,e2) =>
    (case (tc env e1, tc env e2) of
      (Zero_t,t2 as (Neg_t|Pos_t|Zero_t|Int_t) =>
t2
      | (t1 as (Neg_t|Pos_t|Zero_t|Int_t),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., $\text{Pos_t} \leq \text{Int_t}$ 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?
 - $(T_1 * T_2) \leq (U_1 * U_2)$ when $T_1 \leq U_1$ and $T_2 \leq U_2$
 - But only when immutable!
 - Why?
- What about functions?
 - $(T_1 \rightarrow T_2) \leq (U_1 \rightarrow U_2)$ when $U_1 \leq T_1$ and $T_2 \leq U_2$.
 - Why?

Mutability:

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

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:

- $(T_1 * \dots * T_n * T_{n+1}) \leq (U_1 * \dots * U_n)$

when $T_i \leq U_i$.

– Why?

- Non-Permutable Records:

$$\{l_1:T_1, \dots, l_n:T_n, l_{n+1}:T_{n+1}\} \leq \{l_1:U_1, \dots, l_n:U_n\}$$

when $T_i \leq U_i$.

– Assumes $\{x:\text{int}, y:\text{int}\} \neq \{y:\text{int}, x:\text{int}\}$

– That is, the position of a label is independent of the rest of the labels in the type.

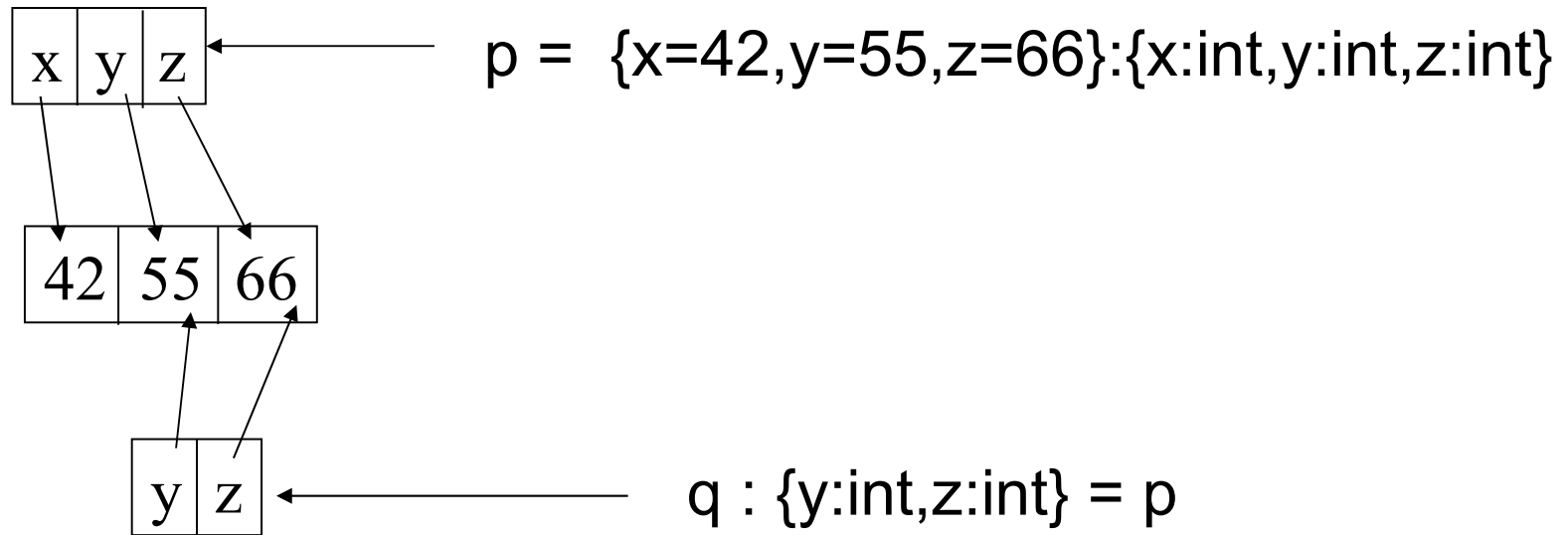
– In ML (or for Java interfaces) this is *not* the case.

ML-Style Records:

- Compiler sorts by label.
- So if you write $\{y:\text{int},z:\text{int},x:\text{int}\}$, the compiler immediately rewrites it to $\{x:\text{int},y:\text{int},z:\text{int}\}$.
- So you need to know all of the labels to determine their positions.
- Consider: $\{y:\text{int},z:\text{int},x:\text{int}\} \leq \{y:\text{int},z:\text{int}\}$
but $\{y,z,x\} == \{x,y,z\} \not\leq \{y,z\}$

If you want both:

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



Checking vs. Inference

When type-checking, we relied upon functions being annotated with the type of their arguments.

```
fun tc (env:var->type) (e:exp) =  
  case e of  
    Var x => env x  
    ...  
  | Lambda(x,t,e) =>  
    Fn_t(t,tc (extend env x t) e)
```

We can infer the result type with this information.
But in ML, you don't even have to supply the argument type!

Type Inference

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

Extend Types with Guesses:

```
datatype 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 val t = guess()
  in
    Fn_t(t,tc (extend env x t) e)
  end

| App(e1,e2) =>
  (case (tc env e1, tc env e2) of
    (Fn_t(t1,t2),t) =>
      if t1 != t then error() else t2
  | (Guess (r as ref None),t) =>
    let val t2 = guess() in
      r := Some(Fn_t(t,t2)); t2
    end
  | (Guess (ref Some (Fn_t(t1,t2))), t) =>
    if t1 != t then error() else t2
```

Cleaner:

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

Where:

```
fun unify (t1:tipe, t2:tipe):bool =
  if (t1 = t2) then true else
  case (t1,t2) of
    (Guess(ref(Some t1')),_) => unify(t1',t2)
  | (Guess(r as (ref None)),t2) =>
      (r := Some t2; true)
  | (_,Guess(_)) => unify(t2,t1)
  | (Int_t,Int_t) => true
  | (Fn_t(t1a,t1b),Fn_t(t2a,t2b)) =>
      unify(t1a,t2a) andalso unify(t1b,t2b)
  | (_,_) => false
```

Subtlety

- Consider: `fn 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:

```
fun unify (t1:tipe, t2:tipe):bool =  
  if (t1 = t2) then true else  
  case (t1,t2) of  
    (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: `fn 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:

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

Naïve ML Type Inference:

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

====>

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

Better Approach (DM):

```
type tvar = string
```

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

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

ML Type Inference

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

Instantiation

```
fun 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:

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

Explanation:

- Each let-bound value is generalized.
 - e.g., $g \rightarrow g$ generalizes to $\text{Forall } a.a \rightarrow a$.
- Each use of a let-bound variable is instantiated with fresh guesses:
 - e.g., if $f:\text{Forall } a.a \rightarrow a$, then in $f e$, then the type we assign to f is $g \rightarrow g$ 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 $g_1 \rightarrow g_2$ and $y:g_1$, 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 val id = (print "Hello"; fn x => x)
in
  (id 42, id "fred")
end
```

!=

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

Problem:

```
let val 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) *)
end
```

"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:

```
fun tc (env:var->type_scheme) (e:exp) =  
  case e of  
    ...  
  | Let(x,e1,e2) =>  
    let val 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:

```
fun may_have_effects e =  
  case e of  
    Int _ => false  
  | Var _ => false  
  | Lambda _ => false  
  | Pair(e1,e2) =>  
    may_have_effects e1 orelse  
    may_have_effects e2  
  | App _ => true
```