HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 14: Type Checking

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

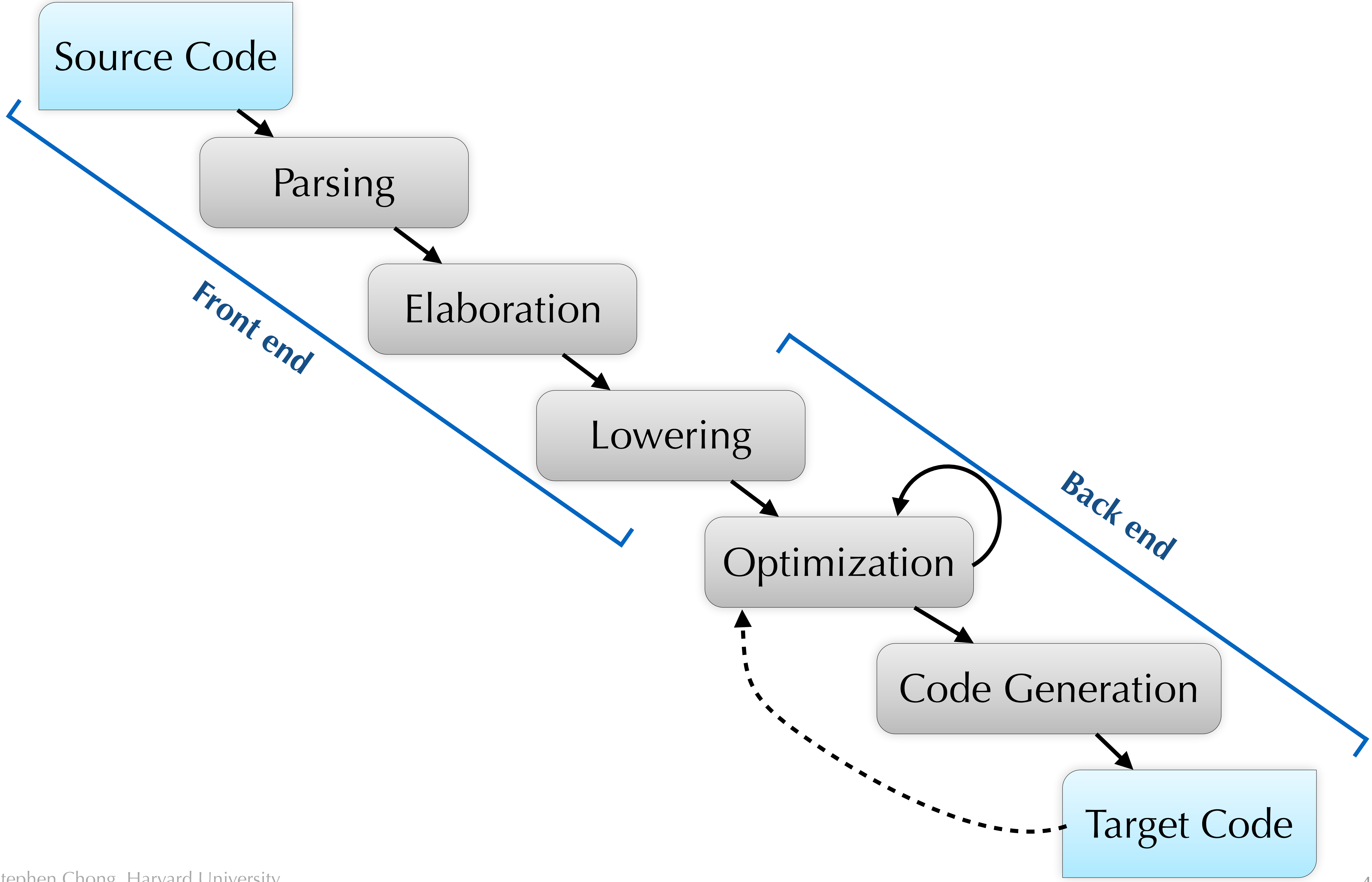
Announcements

- Project 4 out
  - Due Thursday Oct 25 (7 days)
- Project 5 out
  - Due Tuesday Nov 13 (26 days)
- Project 6 will be released Tuesday

Today

- Type checking
- Type inference

Basic Architecture



Source Code

Parsing

Elaboration

Lowering

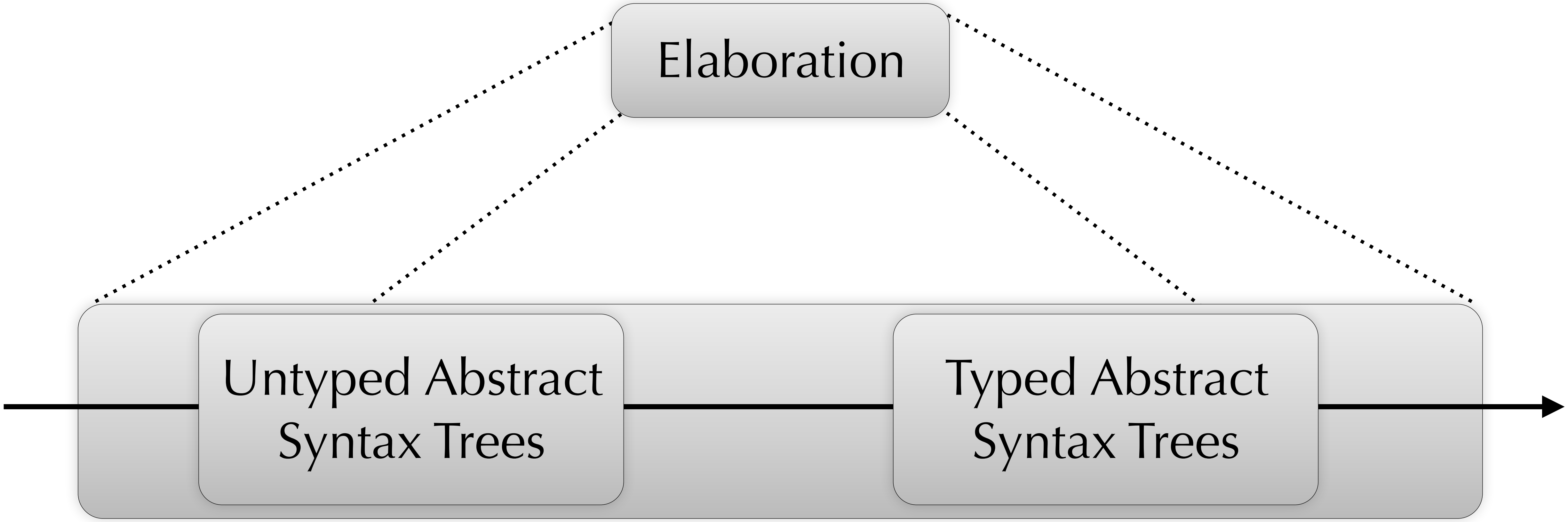
Optimization

Code Generation

Target Code

Front end

Back end



Elaboration

Untyped Abstract Syntax Trees

Typed Abstract Syntax Trees

Undefined Programs

- After parsing, we have AST
- We can interpret AST, or compile it and execute
- But: not all programs are well defined
  - E.g., `3/0`, `“hello” - 7`, `42(19)`
- **Types** allow us to rule out many of these undefined behaviors
  - Types can be thought of as an approximation of a computation
  - E.g., if expression `e` has type `int`, then it means that `e` will evaluate to some integer value
  - E.g., we can ensure we never treat an integer value as if it were a function

What do we do about other operations that our types don’t rule out? e.g., `3/0`

Type Soundness

- Key idea: a well-typed program when executed does not attempt any undefined operation
- Make a model of the source language
  - i.e., an interpreter, or other semantics
  - This tells us were operations are partial
  - Partiality is different for different languages
    - E.g., `“Hi” + “ world”` and `“na”*16` may be meaningful in some languages
- Construct a function to check types: `tc : AST -> bool`
  - AST includes types (or type annotations)
  - If `tc e` returns true, then interpreting `e` will not result in an undefined operation
- Prove that `tc` is correct

Simple Language

```
type tipe =
   Int_t
 | Arrow_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
```

Note: function arguments have type annotation

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_v i, Int_v 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) with
        | 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) -> Arrow_t(t,tc (extend env x t) e)
  | App(e1,e2) ->
    (match (tc env e1, tc env e2) with
       | Arrow_t(t1,t2), t ->
           if (t1 != t) then error() else t2
       | _,_ -> error())
```

Notes

- Type checker is almost like an approximation of the interpreter!
  - But interpreter evaluates function body only when function applied
  - Type checker always checks body of function
- We needed to assume the input of a function had some type `t`$_1$, and reflect this in type of function (`t`$_1$`->t`$_2$)
- At call site (`e`$_1$ `e`$_2$), we don't know what closure e$_1$ will evaluate to, but can calculate type of `e`$_1$ and check that `e`$_2$ has type of argument

Growing the Language

- Adding booleans...

```
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())
```

Type Inference

- Type checking is great if we already have enough type annotations
  - For our simple functional language, sufficed to have type annotations for function arguments
- But what about if we tried to infer types?
- Key idea: we will “guess” each missing type annotation, and update our guess based on how the program uses that function and function argument

```
let rec tc (env:(var*tipe) list) (e:exp) =
  match e with
  | Lambda(x,e) ->
      (let t = guess() in
          Arrow_t(t,tc (extend env x t) e))
```

Extend Types with Guesses

- A guess represents an initially unknown type
  - Type inference will update the type as it gets more information

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

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

Must Handle Guesses

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

Cleaner Version...

```
let rec tc (env: (var*tipe) list) (e:exp) =
  match e with
   | Var x -> lookup env x
   | Lambda(x,e) ->
       let t = guess() in
             Arrow_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 (Arrow_t(t2,t)) then t
        else error()
```

Unification

```
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 := Some t2; true)
  | _, Guess(_) -> unify t2 t1
  | Int_t, Int_t -> true
  | Arrow_t(t1a,t1b), Arrow_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,Arrow_t(g1,g2))`
  - So we set `g1 := Some(Arrow_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
  case (t1,t2) of
    (Guess(r),_) when !r = None ->
     if occurs r t2 then error()
     else (r := Some t2; true)
  | ...
```

- Alternatively, be careful not to loop anywhere.
  - In particular, when considering the cases for `(t1, t2)`, make sure it doesn’t go into an infinite loop

Polymorphism

- Consider: `fun x -> x`
- We guess `g1` for `x`
  - We see `x`
  - So `g1` is the result.
  - We return `Arrow_t(g1,g1)`
  - `g1` is unconstrained
  - We could constraint it to `Int_t` or `Arrow_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


let f = fun x -> x in (f 3, f “foo”)
```

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
        Arrow_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
```

is type checked as if it were

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

Effects

- But this can be inefficient!
- And in a type system that considers effects, does not accurately reflect how the program executes

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

is not equivalent to

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

Hindley-Milner Type Inference

- **Polymorphism** is the ability of code to be used on values of different types.
  - E.g., polymorphic function can be invoked with arguments of different types
  - Polymorph means “many forms”
- OCaml has polymorphic types
  - e.g., `val swap : 'a ref -> 'a -> ‘a = ...`
- But type inference for full polymorphic types is undecidable...
- OCaml has restricted form of polymorphism that allows type inference: **let-polymorphism** aka prenex polymorphism
  - Allow let expressions to be typed polymorphically, i.e., used at many types
  - Doesn’t require copying of let expressions
  - Requires clear distinction between polymorphic types and non-polymorphic types...

Hindley-Milner Type Inference

```
type tvar = string

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

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

Type variables are placeholders for types

Type schemes are polymorphic types

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 g = guess() in
    Arrow_t(g,tc (extend env x (Forall([],g)) 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 =
  match s with
  | Forall(vs,t) ->
    let b = map (fn a -> (a,guess()) vs in
    substitute(b,t)
```

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

- Every variable in environment maps to a type scheme, i.e., universally quantified type, possibly with empty list of quantifiers
- Each let-bound variable is generalized
  - E.g., `g->g` generalizes to `Forall a. a->a`
- Each use of let-bound variable is instantiated with fresh guesses
  - E.g., if `f:Forall a. a->a`, then if `f e`, we instantiate the type of `f` to `g->g` for some fresh guess `g`
- But only generalize variables that appear only in let and not in environment
  - Variables in environment may be later constrained, e.g., `function f(y) = let g = fn x -> (x, y) in (g y + 7)`
  - In expression `fn x -> (x, y)` can generalize for type of `x`, but not for type of `y`

Difficulties with Mutability

```
let r = ref (fun x -> x)
    (* r : Forall 'a: ref('a->'a)  *)
in
  r := (fun 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
```