CS 515: Programming Languages and Compiler II

Type Checking

Micro-OCaml Expression Grammar

e ::= x | n | e + e | `let` x `=` e `in` e

- ***e***, ***x***, ***n*** are *meta-variables* that stand for categories of syntax
  - ***x*** is any identifier (like `z`, `y`, `foo`)
  - ***n*** is any numeral (like `1`, `0`, `10`, `-25`)
  - ***e*** is any expression (here defined, recursively!)

- *Concrete syntax* of actual expressions in **black**
  - Such as `let`, `+`, `z`, `foo`, `in`, …

- ::= and | are *meta-syntax* used to define the syntax of a language (part of "Backus-Naur form," or BNF)

Micro-OCaml Expression Grammar

$$e ::= x \mid n \mid e + e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e$$

Examples

- `1` is a numeral *n* which is an expression *e*
- `1+z` is an expression *e* because
  - `1` is an expression *e*,
  - `z` is an identifier *x*, which is an expression *e*, and
  - *e* + *e* is an expression *e*
- `let z = 1 in 1+z` is an expression *e* because
  - `z` is an identifier *x*,
  - `1` is an expression *e* ,
  - `1+z` is an expression *e*, and
  - `let` *x* `=` *e* `in` *e* is an expression *e*

Abstract Syntax = Structure

- Here, the grammar for ***e*** is describing its abstract syntax tree (AST), i.e., ***e***'s structure

e ::= ***x*** | ***n*** | **e** + **e** | **let** ***x*** **=** **e** **in** **e**

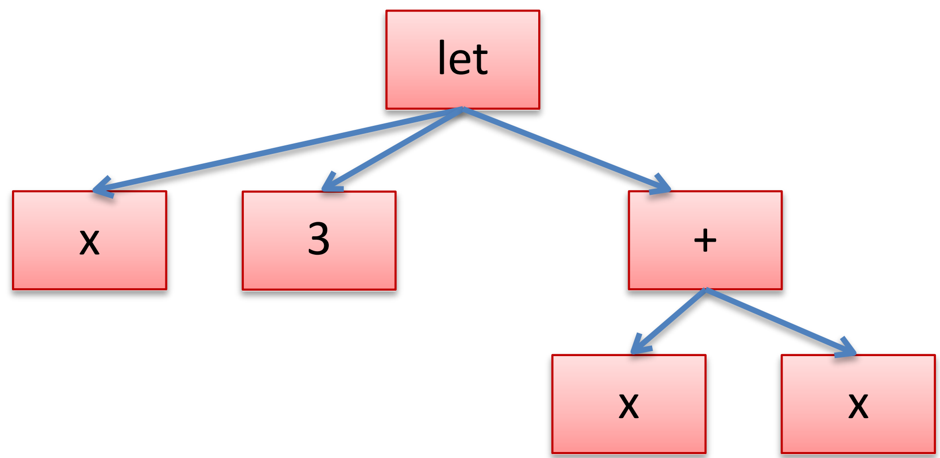
corresponds to (in definitional interpreter)

```
type id = string
type num = int
type exp =
    | Ident of id                  (* x *)
    | Num of num                   (* n *)
    | Plus of exp * exp          (* e+e *)
    | Let of id * exp * exp
                           (* let x=e in e *)
```

Representing Syntax

- Program syntax is a complicated tree-like data structure

```
let x = 3 in
x + x
```



Representing Syntax

We can represent the OCaml program:

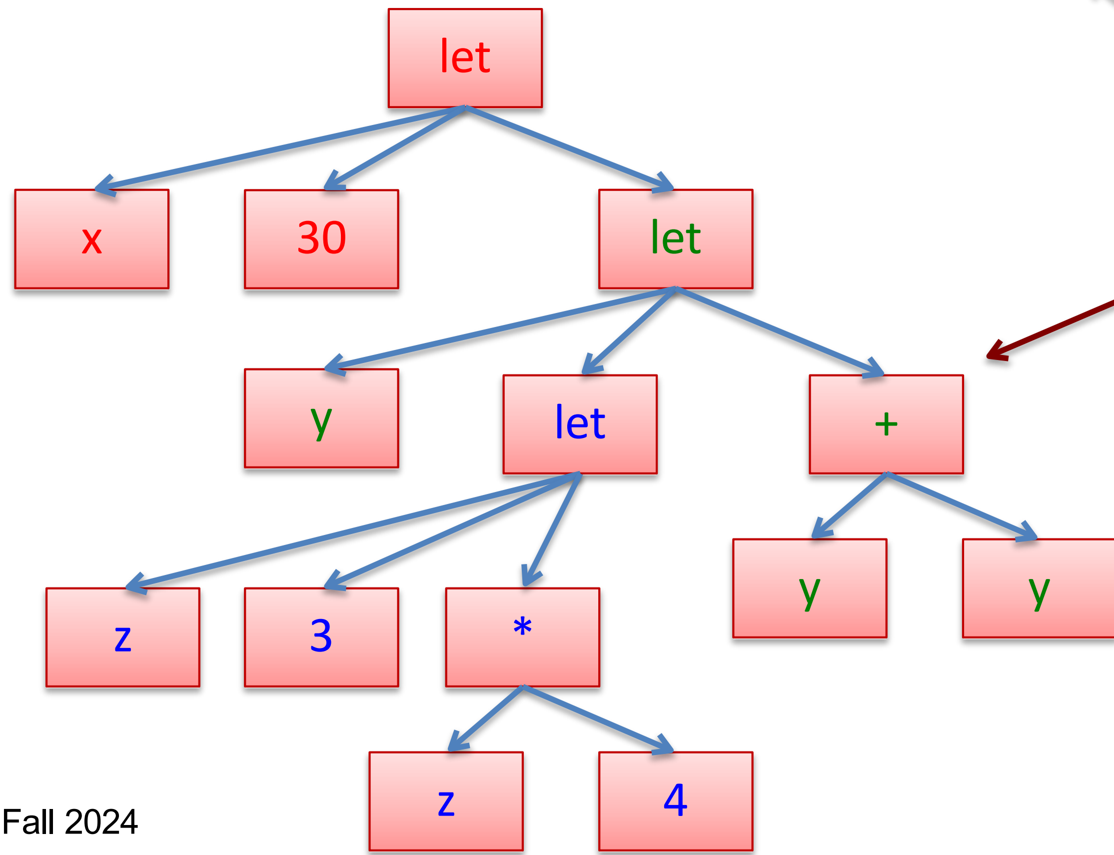
```
let x = 30 in
  let y =
    (let z = 3 in
     z+4)
  in
  y+y
```

as an exp value:

```
    Let("x", Num 30,
          Let("y",
                Let("z", Num 3,
                      Plus(Ident "z", Num 4)),
          Plus(Ident "y", Ident "y")
```

Representing Syntax

```
Let("x",Num 30,
      Let("y",Let("z",Num 3,
                      Plus(Ident "z", Num 4)),
           Plus(Ident "y", Ident "y")
```



Notice how the OCaml expression can be drawn as a tree

Values

- An expression's final result is a value. What can values be?

$$v ::= n$$

- Just numerals for now
  - In terms of an interpreter's representation:

```
type value = int
```

  - In a full language, values v will also include booleans (**`true`**, **`false`**), strings, functions, …

Interpreter

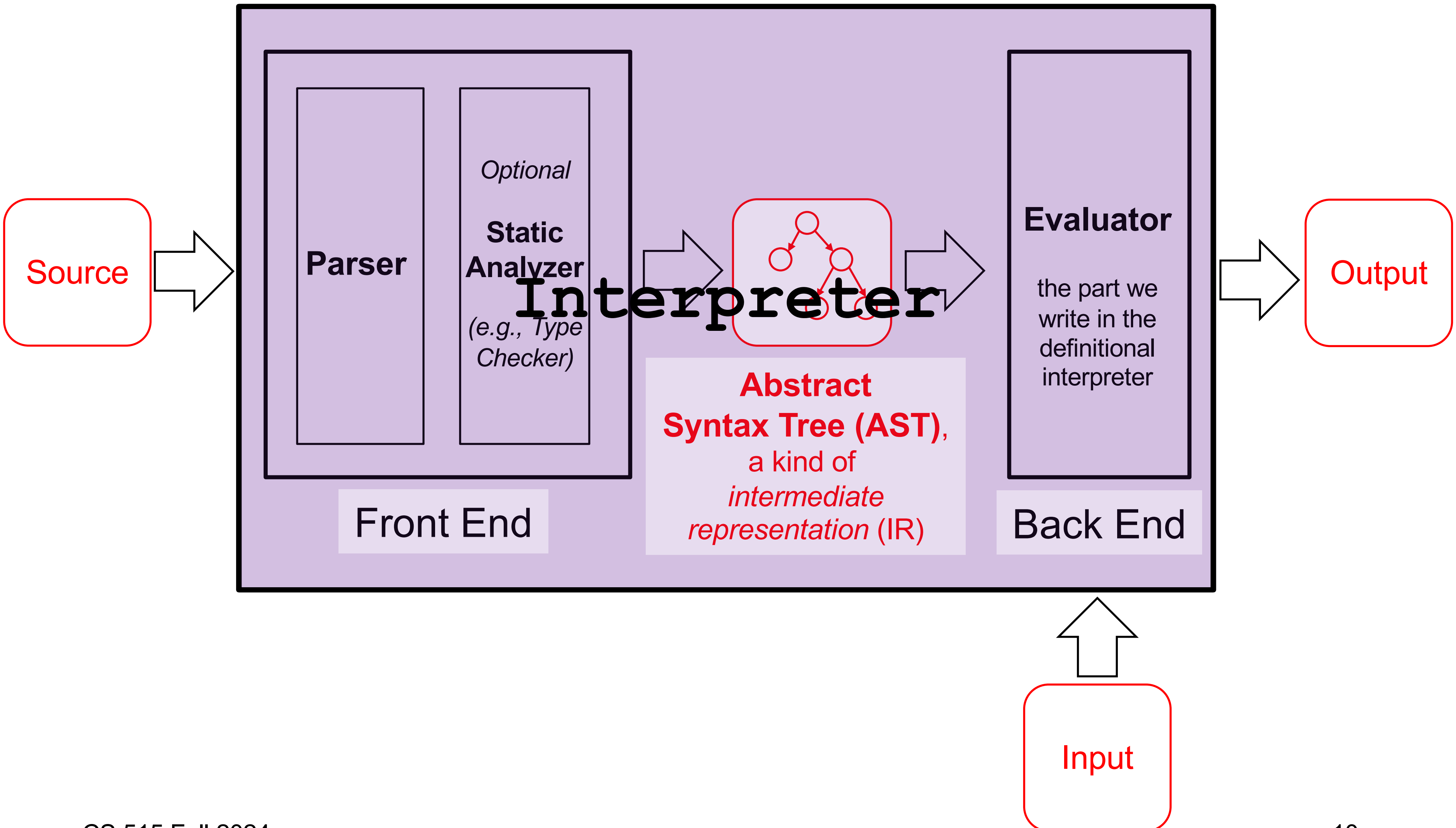
The semantics is represented as a function

```
eval: exp -> value
```

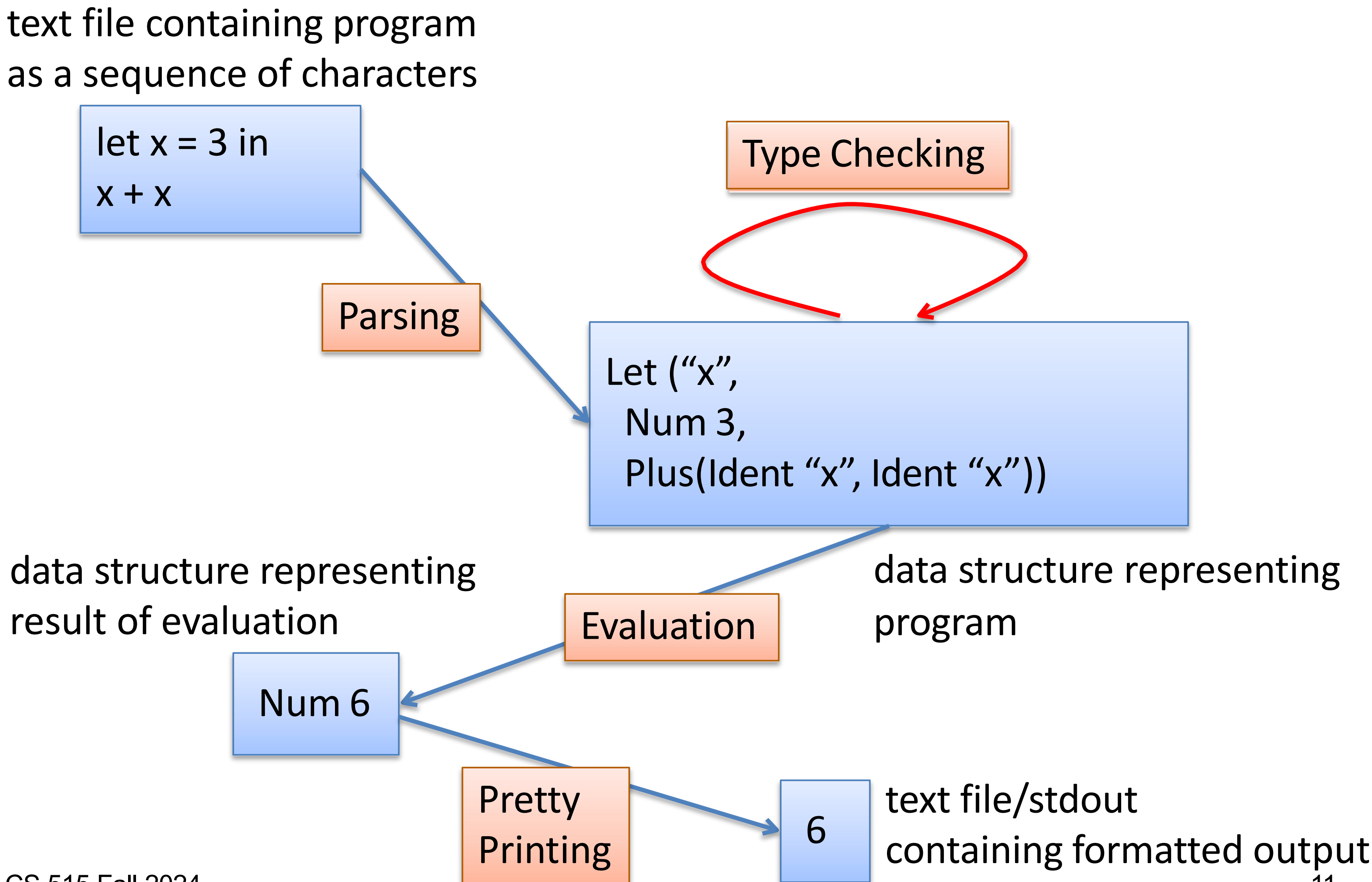
```
type id = string
type num = int
type exp =
    | Ident of id                 (* x *)
    | Num of num                  (* n *)
    | Plus of exp * exp         (* e+e *)
    | Let of id * exp * exp
                         (* let x=e in e *)
```

```
type value = int
```

Aside: Real Interpreters



Example: Real Interpreters



Quick Look: Type Checking

- Inference rules to specify a program's static semantics
  - I.e., the rules for type checking

- Types t ::= `bool` | `int` | `t->t`
- Judgment $\vdash e : t$ says e has type t
  - We define inference rules for this judgment

Rules of Inference

- We can use a more compact notation for the rules we just presented: rules of inference
  - Has the following format

$$\frac{H_1 \quad \ldots \quad H_n}{C}$$

  - Says: if the conditions $H_1 \quad \ldots \quad H_n$ ("hypotheses") are true, then the condition C ("conclusion") is true
  - If n=0 (no hypotheses) then the conclusion automatically holds; this is called an axiom

Some Type Checking Rules

- Boolean constants have type **`bool`**

$$\frac{}{\vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}}$$

- Equality checking has type **`bool`** too
  - Assuming its target expression has type **`int`**

$$\frac{\vdash e : \texttt{int}}{\vdash \texttt{eq0}\ e : \texttt{bool}}$$

- Conditionals

$$\frac{\vdash e1 : \texttt{bool} \qquad \vdash e2 : t \qquad \vdash e3 : t}{\vdash \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 : t}$$

Handling Binding

- What about the types of variables?
  - Taking inspiration from the environment-style operational semantics, what could you do?

- Change judgment to be $G \vdash e : t$ which says e *has type* t *under type environment* G
  - G is a map from variables x to types t
    - It maps vars to types

- What would be the rules for **`let`**, and variables?

Type Checking with Binding

- Variable lookup

$$\frac{G(x) = t}{G \vdash x : t}$$

- Let binding

$$\frac{G \vdash e1 : t1 \qquad G, x : t1 \vdash e2 : t2}{G \vdash \texttt{let } x = e1 \texttt{ in } e2 : t2}$$

Type Checking Rules

```
t ::= bool | int | t->t

e ::=
  x
  | n
  | true
  | false
  | e o e
  | let x = e in e
  | if e then e else e
  | fun x:t -> e
  | e e

o ::= + |-| <
```

Goal: Give rules that define the relation "G ⊢ e : t".

To do that, we are going to give one rule for every sort of expression.

(We can turn each rule into a case of a recursive function that implements it pretty directly.)

Type Checking Rules

```
t ::= bool | int | t->t

e ::=
   x
   | n
   | true
   | false
   | e o e
   | let x = e in e
   | if e then e else e
   | fun x:t -> e
   | e e

o ::= + |-| <
```

Rule for constant integers:

$$\frac{}{G \vdash n : int}$$

English:

"integer constants n *always* have type int, no matter what the context G is"

Type Checking Rules

```
t ::= bool | int | t->t

e ::=
   x
   | n
   | true
   | false
   | e o e
   | let x = e in e
   | if e then e else e
   | fun x:t -> e
   | e e

o ::= + |-| <
```

Rule for constant booleans:

$$\frac{}{G \vdash \text{true} : \text{bool}} \qquad \frac{}{G \vdash \text{false} : \text{bool}}$$

English:

"boolean constants b *always* have type bool, no matter what the context G is"

Type Checking Rules

```
t ::= bool | int | t->t

e ::=
  x
  | n
  | true
  | false
  | e o e
  | let x = e in e
  | if e then e else e
  | fun x:t -> e
  | e e

o ::= + |-| <
```

Rule for operators:

$$\frac{G \vdash e1 : t1 \qquad G \vdash e2 : t2 \qquad optype(o) = (t1, t2, t3)}{G \vdash e1\ o\ e2 : t3}$$

where

optype (+) = (int, int, int)
optype (-) = (int, int, int)
optype (<) = (int, int, bool)

English:

> “e1 o e2 has type t3, if e1 has type t1, e2 has type t2 and o is an operator that takes arguments of type t1 and t2 and returns a value of type t3"

Type Checking Rules

```
t ::= bool | int | t->t

e ::=
   x
   | n
   | true
   | false
   | e o e
   | let x = e in e
   | if e then else e
   | fun x:t -> e
   | e e

o ::= + |-| <
```

Rule for variables:

look up x in context G

$$\frac{}{G \vdash x : G(x)}$$

English:

"variable x has the type given by the context"

Note: this is rule explains (part) of why the context needs to provide types for all of the free variables in an expression

Type Checking Rules

```
t ::= bool | int | t->t

e ::=
   x
   | n
   | true
   | false
   | e o e
   | let x = e in e
   | if e then e else e
   | fun x:t -> e
   | e e

o ::= + |-| <
```

Rule for if:

$$\frac{G \vdash e1 : bool \qquad G \vdash e2 : t \qquad G \vdash e3 : t}{G \vdash \text{if } e1 \text{ then } e2 \text{ else } e3 : t}$$

English:

“if e1 has type bool
and e2 has type t
and e3 has (the same) type t
then e1 then e2 else e3 has type t "

Type Checking Rules

```
t ::= bool | int | t->t

e ::=
   x
   | n
   | true
   | false
   | e o e
   | let x = e in e
   | if e then e else e
   | fun x:t -> e
   | e e

o ::= + |-| <
```

Rule for functions:

To know how to extend the context G, we need the typing annotation on the argument

$$\frac{G, x{:}t \vdash e : t2}{G \vdash (\text{fun } x{:}t \text{ -> } e) : t \text{ -> } t2}$$

English:

"if G extended with x:t proves e has type t2 then funx:t -> e has type t -> t2 "

Type Checking Rules

```
t ::= bool | int | t->t

e ::=
   x
   | n
   | true
   | false
   | e o e
   | let x = e in e
   | if e then e else e
   | fun x:t -> e
   | e e

o ::= + |-| <
```

Rule for function call:

$$\frac{G \vdash e1 : t1 \to t2 \qquad G \vdash e2 : t1}{G \vdash e1\ e2 : t2}$$

English:

> “if e1 has type t1->t2 and e2 has type t1 then e1 e2 has type t2 "

Type Checking Rules

```
t ::= bool | int | t->t

e ::=
   x
   | n
   | true
   | false
   | e o e
   | let x = e in e
   | if e then e else e
   | fun x:t -> e
   | e e

o ::= + |-| <
```

Rule for let:

$$\frac{G \vdash e1 : t1 \qquad G,x{:}t1 \vdash e2 : t2}{G \vdash \text{let } x = e1 \text{ in } e2 : t2}$$

English:

"if e1 has type t1
and G extended with x:t1 proves e2 has
type t2 then let x = e1 in e2 has type t2 "

Key Properties

- Good type systems are sound.
  - ie, well-typed programs have "well-defined" evaluation
  - An interpreter should not raise an exception part-way through because it doesn't know how to continue evaluation
- Examples of OCaml expressions that go wrong:
  - true + 3 (addition of booleans not defined)
  - let (x,y) = 17 in ... (can’t extract fields of int)
  - true (17) (can’t use a bool as if it is a function)

Soundness = Progress + Preservation

- Sound type systems *accurately* predict run time behavior
  - if e : int and e terminates then e evaluates to an integer

- Progress Theorem:
  - If ⊢ e : t then either:
  - (1) e is a value, or (2) e --> e’

- Preservation Theorem:
  - If ⊢ e : t and e --> e' then ⊢ e' : t

Type Checking

- The typing rules also define an algorithm for type checking:
  - If you view G and e as inputs,
  - the rules for “G ⊢ e : t” tell you how to compute t

The Syntax

```
type t = IntT                        (* type int *)
       | BoolT                       (* type bool *)
       | ArrT of t * t               (* type t -> t *)


type o = Plus | Minus | LessThan     (* operators *)

type e =                             (* expressions *)
  Int of int
  | Bool of bool
  | Fun of string * t * e            (* t gives type of argument *)
  | Ident of string
  | Op of e * o * e
  | If of e * e * e
  | Let of string * e * e
  | App of e * e
```

Context Operation

```
(* abstract type of contexts *)
type ctx = (id * t) list

(* update ctx x t: updates context ctx by binding variable x to type t *)
let extend ctx x ty = (x,ty)::ctx


(* look ctx x: retrieves the type t associated with x in ctx
 *            raises NotFound if x does not appear in ctx *)
let rec lookup ctx x =
  match ctx with
    [] -> raise NotFound
  | (y,ty)::ctx' ->
    if x = y then ty
    else lookup ctx' x
```

Built-in Functions

- The types for library functions must be provided.

```
(* op o = (t1, t2, t3) when o has type t1 -> t2 -> t3 *)
let op (o : o) : t =
 match o with
 | Plus -> (IntT, IntT, IntT)
 | Minus -> (IntT, IntT, IntT)
 | LessThan -> (IntT,IntT,Bool)
```

Simple Rules

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with
  | Int v -> IntT
  | Bool v -> BoolT

  | Op (e1, o, e2) ->
      let (t1, t2, t) = op o in     (* op : t1 -> t2 -> t3 *)
      let t1' = check ctx e1 in
      let t2' = check ctx e2 in
      if (t1 = t1') && (t2 = t2') then t3
      else
        failwith "bad argument to operator"
```

$$\frac{}{G \vdash \text{true} : \text{bool}}$$

$$\frac{}{G \vdash \text{false} : \text{bool}}$$

$$\frac{}{G \vdash n : \text{int}}$$

$$\frac{\text{optype}(o) = (t1, t2, t3) \quad G \vdash e1 : t1 \quad G \vdash e2 : t2}{G \vdash e1 \text{ o } e2 : t3}$$

Simple Rules

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

   | Ident x ->
          (try look ctx x with
            NotFound -> failwith ("free variable: " ^ x))
```

$$\frac{}{G \vdash x : G(x)}$$

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | Fun (x,t,e) ->
       check (update ctx x t) e
```

$$\frac{G, x{:}t \vdash e : t2}{G \vdash (\text{fun } x{:}t \text{ -> } e) : t \text{ -> } t2}$$

Notice that if we did not have the type t as a typing annotation we would not be able to make progress in our type checker at this point. We need to have a type for the variable x in our context in order to recursively check the expression e

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | Fun (x,t,e) ->
       check (update ctx x t) e
```

$$\frac{G, x{:}t \vdash e : t2}{G \vdash (\text{fun } x{:}t \text{ -> } e) : t \text{ -> } t2}$$

```
let f =
  fun (x:int) -> x + 1
in
f 10

ctx = {}
```

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | Fun (x,t,e) ->
      check (update ctx x t) e
```

$$\frac{G, x{:}t \vdash e : t2}{G \vdash (\text{fun } x{:}t \text{ -> } e) : t \text{ -> } t2}$$

```
let f =
  fun (x:int) -> x + 1
in
f 10

ctx = {x:int}
```

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | Fun (x,t,e) ->
      check (update ctx x t) e
```

$$\frac{G, x{:}t \vdash e : t2}{G \vdash (\text{fun } x{:}t \text{ -> } e) : t \text{ -> } t2}$$

```
let f =
  fun (x:int) -> x + 1
in
f 10

ctx = {x:int}
```

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | App (e1, e2) ->
      begin
        let t1 = check ctx e1 in
        match t1 with
         | ArrT (targ, tresult) ->
             let t2 = check ctx e2 in
             if targ = t2 then tresult
             else failwith "bad argument to function"
        | _ -> failwith "not a function in call site"
      end
```

$$\frac{G \vdash e1 : targ \rightarrow tresult \qquad G \vdash e2 : targ}{G \vdash e1\ e2 : tresult}$$

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | App (e1, e2) ->
      begin
        let t1 = check ctx e1 in
        match t1 with
         | ArrT (targ, tresult) ->
             let t2 = check ctx e2 in
             if targ = t2 then tresult
             else failwith "bad argument to function"
        | _ -> failwith "not a function in call site"
      end
```

$$\frac{G \vdash e1 : targ \rightarrow tresult \qquad G \vdash e2 : targ}{G \vdash e1\ e2 : tresult}$$

```
let f =
  fun (x:int) -> x + 1
in
f 10

ctx = {f:int->int}
```

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | App (e1, e2) ->
      begin
        let t1 = check ctx e1 in
        match t1 with
         | ArrT (targ, tresult) ->
             let t2 = check ctx e2 in
             if targ = t2 then tresult
             else failwith "bad argument to function"
        | _ -> failwith "not a function in call site"
      end
```

$$\frac{G \vdash e1 : targ \rightarrow tresult \qquad G \vdash e2 : targ}{G \vdash e1\ e2 : tresult}$$

```
let f =
  fun (x:int) -> x + 1
in
f 10

ctx = {f:int->int}
```

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | App (e1, e2) ->
      begin
        let t1 = check ctx e1 in
        match t1 with
         | ArrT (targ, tresult) ->
             let t2 = check ctx e2 in
              if targ = t2 then tresult
              else failwith "bad argument to function"
        | _ -> failwith "not a function in call site"
      end
```

$$\frac{G \vdash e1 : targ \to tresult \qquad G \vdash e2 : targ}{G \vdash e1\ e2 : tresult}$$

```
let f =
  fun (x:int) -> x + 1
in
f 10

ctx = {f:int->int}
```

Function Typing

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | App (e1, e2) ->
      begin
        let t1 = check ctx e1 in
        match t1 with
         | ArrT (targ, tresult) ->
             let t2 = check ctx e2 in
             if targ = t2 then tresult
             else failwith "bad argument to function"
        | _ -> failwith "not a function in call site"
      end
```

$$\frac{G \vdash e1 : targ \rightarrow tresult \qquad G \vdash e2 : targ}{G \vdash e1\ e2 : tresult}$$

```
let f =
  fun (x:int) -> x + 1
in
f 10

ctx = {f:int->int}
```

Exercise: Other Rules

```
(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with

  | If (e1, e2, e3) -> ...

  | Let (x, e1, e2) -> ...
```

Review: Type Checking

A function check : context -> exp -> type

- requires function arguments to be annotated with types
- specified using formal rules. eg, the rule for function call:

```
let f =
  fun (x:int) -> x + 1 in
f 10
```

$$\frac{G \mid\!- e1 : t1 \rightarrow t2 \qquad G \mid\!- e2 : t1}{G \mid\!- e1\ e2 : t2}$$

Type Schemes

A *type scheme* contains type variables that may be filled in during type inference

s ::= *a* | int | bool | s -> s

A *term scheme* is a term that contains type schemes rather than proper types. eg, for functions:

fun (x:s) -> e

let rec f (x:s) : s = e

Main Algorithm

- Add distinct variables in all places type schemes are needed
- Generate constraint (equations between types) that must be satisfied in order for an expression to type check
- Solve the equations, generating substitutions of types for the variables.

Example: Inferring types for map

```
let rec map f l =
    match l with
        [] -> []
    | hd::tl -> f hd :: map f tl
```

Step 1: Annotate

constraints

```
let rec map (f:a)  (l:b)  : c =
    match l with
        [] -> []
    | hd::tl -> f hd :: map f tl
```

type schemes
on functions

Step 2: Generate Constraints

b = b′ list

```
let rec map (f:a) (l:b) : c =
    match l with
        [] -> []
       |(hd:b')::(tl:b' list) ->
         f hd :: map f tl
```

constraints
b = b′ list

Step 2: Generate Constraints

b = b′ list

```
let rec map (f:a) (l:b) : c =
    match l with
        [] -> []
        |(hd:b′)::(tl:b′ list) ->
            f hd :: map f tl
```

a = a   b = b′ list

constraints
b = b′ list
a = a
b = b′ list

Step 2: Generate Constraints

b = b′ list

constraints
b = b′ list
a = a
b = b′ list
a = b′ -> a′

```
let rec map (f:a) (l:b) : c =
    match l with
        [] -> []
        |(hd:b′)::(tl:b′ list) ->
      (f (hd:b′)):a′ :: map f tl
```

a = b′ -> a′ a = a b = b′ list

Step 2: Generate Constraints

b = b′ list

```
let rec map (f:a) (l:b) : c =
    match l with
        [] -> []
        | (hd:b′)::(tl:b′ list) ->
    ((f (hd:b′)):a′ :: (map f tl):c):c′ list
```

constraints
b = b′ list
a = a
b = b′ list
a = b′ -> a′
c = c′ list
a′ = c′

a = b′ -> a′

a = a

b = b′ list

c = c′ list
a′ = c′

Step 2: Generate Constraints

b = b′ list

```
let rec map (f:a) (l:b) : c =
    match l with
      [] -> [] :d list
      | (hd:b′)::(tl:b′ list) ->
   ((f (hd:b′)):a′::(map f tl):c):c′ list
```

constraints
b = b′ list
a = a
b = b′ list
a = b′ -> a′
c = c′ list
a′ = c′
d list = c′ list

a = b′ -> a′

a = a   b = b′ list

d list = c′ list

c = c′ list
a′ = c′

Step 2: Generate Constraints

b = b′ list

```
let rec map (f:a) (l:b) : c =
     match l with
        [] -> [] :d list
        | (hd:b′)::(tl:b′ list) ->
   ((f (hd:b′)):a′ ::(map f tl):c):c′ list
```

constraints
b = b′ list
a = a
b = b′ list
a = b′ -> a′
c = c′ list
a′ = c′
d list = c′ list
d list = c
c′ list = c

a = b′ -> a′

a = a   b = b′ list

d list = c′ list

c = c′ list
a′ = c′

d list = c
c′ list = c

Step 3: Solve Constraints

```
let rec map (f:a) (l:b) : c =
    match l with
        [] -> []
    | hd::tl -> f hd :: map f tl
```

constraints

b = b′ list
a = a
b = b′ list
a = b′ -> a′
c = c′ list
a′ = c′
d list = c′ list
d list = c
c′ list = c

solution

```
[b' -> c'/a]
[b' list/b]
[c' list/c]
```

Step 3: Solve Constraints

```
let rec map (f:a) (l:b) : c =
    match l with
        [] -> []
    | hd::tl -> f hd :: map f tl
```

final solution:

```
[b' -> c'/a]
[b' list/b]
[c' list/c]
```

```
let rec map (f:b' -> c') (l:b' list) : c' list =
    match l with
        [] -> []
    | hd::tl -> f hd :: map f tl
```

Type Inference Details

- Type constraints are sets of equations between type schemes
  - q ::= {s11 = s12, ..., sn1 = sn2}
  - e.g.: {b = b' list, a = (b -> c)}

Constraint Generation

- Syntax-directed constraint generation
  - our algorithm crawls over abstract syntax of untyped expressions and generates
    - a term scheme
    - a set of constraints

Constraint Generation

- Algorithm defined as set of inference rules:
  - G ⊢ u => e : t, q

constraints that must be solved

annotated expression

unannotated expression

inputs

outputs

in OCaml:

```
gen : ctxt -> exp ->
        ann_exp * scheme * constraints
```

Constraint Generation

- Simple rules:
  - $G \vdash x \Longrightarrow x : s, \{\ \}$ (if $G(x) = s$)
  - $G \vdash n \Longrightarrow n : int, \{\ \}$
  - $G \vdash true \Longrightarrow true : bool, \{\ \}$
  - $G \vdash false \Longrightarrow false : bool, \{\ \}$

Operators

```
G ⊢ u1 ==> e1 : t1, q1            G ⊢ u2 ==> e2 : t2, q2
-----------------------------------------------------------
G ⊢ u1 + u2 ==> e1 + e2 : int, q1 U q2 U {t1 = int, t2 = int}
```

```
G ⊢ u1 ==> e1 : t1, q1            G ⊢ u2 ==> e2 : t2, q2
-----------------------------------------------------------
G ⊢ u1 < u2 ==> e1 < e2 : bool, q1 U q2 U {t1 = int, t2 = int}
```

If Expressions

G ⊢ u1 ==> e1 : t1, q1
G ⊢ u2 ==> e2 : t2, q2
G ⊢ u3 ==> e3 : t3, q3

G ⊢ if u1 then u2 else u3 ==> if e1 then e2 else e3

: t2, q1 U q2 U q3 U {t1=bool, t2 = t3}

Function Application

$$\frac{G \vdash u1 \Longrightarrow e1 : t1,\ q1 \qquad G \vdash u2 \Longrightarrow e2 : t2,\ q2 \qquad \text{(for fresh a)}}{G \vdash u1\ u2 \Longrightarrow e1\ e2 \quad : \quad a, \quad q1 \cup q2 \cup \{t1 = t2 \to a\}}$$

Example

b = b′ list

```
let rec map (f:a) (l:b) : c =
    match l with
        [] -> []
        |(hd:b′)::(tl:b′ list) ->
       (f (hd:b′)):a′ :: map f tl
```

a = b′ -> a′

Function Definition

```
G, x : a ⊢ u ==> e : t, q                        (for fresh a,b)
------------------------------------------------------------------
G ⊢ (fun x -> u) ==> (fun (x  -> e)  : a -> b, q U {t = b}
```

Example

```
let rec map (f:a) (l:b) : c =
    match l with
        [] -> []
    | hd::tl -> f hd :: map f tl
```

type schemes
on functions

Step 2: Generate Constraints

b = b′ list

```
let rec map (f:a) (l:b) : c =
    match l with
       [] -> [] :d list
       | (hd:b′)::(tl:b′ list) ->
  ((f (hd:b′)):a′ ::(map f tl):c):c′ list
```

d list = c′ list

d list = c
c′ list = c

Function Definition

```
G, f : a -> b, x : a ⊢ u ==> e : t, q                    (for fresh a,b)
-------------------------------------------------------------------
G ⊢ (rec f(x) = u) ==> (rec f (x : a) : b = e)  :   a -> b, q U {t = b}
```

Summary: The type inference system

```
G ⊢ u1 ==> e1 : t1, q1          G ⊢ u2 ==> e2 : t2, q2
---------------------------------------------------------------------------
G ⊢ u1 + u2 ==> e1 + e2 : int, q1 U q2 U {t1 = int, t2 = int}
```

```
G ⊢ u1 ==> e1 : t1, q1
G ⊢ u2 ==> e2 : t2, q2
G ⊢ u3 ==> e3 : t3, q3
---------------------------------------------------------------
G ⊢ if u1 then u2 else u3 ==> if e1 then e2 else e3
            : t2,     q1 U q2 U q3 U {t1=bool, t2 = t3}
```

```
G ⊢ x ==> x : s, { }      (if G(x) = s)
```

```
G ⊢ n ==> n : int, { }
```

```
G ⊢ u1 ==> e1 : t1, q1
G ⊢ u2 ==> e2 : t2, q2             (for fresh a)
---------------------------------------------------------------
G ⊢ u1 u2==> e1 e2       :       a,     q1 U q2 U {t1 = t2 -> a}
```

```
G, x : a ⊢ u ==> e : t, q                    (for fresh a)
---------------------------------------------------------------------------
G ⊢ fun x -> u ==> fun (x : a) -> e    :     a -> t,       q
```

```
G, f : a -> b, x : a ⊢ u ==> e : t, q            (for fresh a,b)
---------------------------------------------------------------------
G ⊢ rec f(x) = u ==> rec f (x : a) : b = e     :     a -> b, q U {t = b}
```

Solving Constraints

- A solution to a system of type constraints is a ***substitution S***
  - a function from type variables to types
  - assume substitutions are defined on all type variables:
    - S(a) = a   (for almost all variables a)
    - S(a) = s   (for some type scheme s)

We can also apply a substitution S to a full type scheme s.

apply: [int/a,   int->bool/b]

to: b -> a -> b

returns: (int->bool) -> int -> (int->bool)

Substitutions

- When is a substitution S a solution to a set of constraints?
- Constraints: { s1 = s2, s3 = s4, s5 = s6, ... }
- When the substitution makes both sides of all equations the same.

constraints:

```
a = b -> c
c = int -> bool
```

solution:

```
b -> (int -> bool)/a
int -> bool/c
b/b
```

Substitutions

- When is a substitution S a solution to a set of constraints?
- Constraints: { s1 = s2, s3 = s4, s5 = s6, ... }
- When the substitution makes both sides of all equations the same.

constraints:

```
a = b -> c
c = int -> bool
```

solution:

```
b -> (int -> bool)/a
int -> bool/c
b/b
```

constraints with solution applied:

```
b -> (int -> bool)    =    b -> (int -> bool)
       int -> bool    =    int -> bool
```

Substitutions

- When is a substitution S a solution to a set of constraints?
- Constraints: { s1 = s2, s3 = s4, s5 = s6, ... }
- When the substitution makes both sides of all equations the same.

solution:

b -> (int -> bool)/a
int -> bool/c
b/b

constraints:

a = b -> c
c = int -> bool

solution 2:

int->(int->bool) / a
int->bool / c
int / b

Substitutions

- When is one solution better than another to a set of constraints?

constraints:

```
a = b -> c
c = int -> bool
```

solution 1:

```
b->(int->bool)  /  a
int->bool  /  c
b  /  b
```

type b -> c with solution applied:

```
b -> (int -> bool)
```

solution 2:

```
int->(int->bool)  /  a
int->bool  /  c
int  /  b
```

type b -> c with solution applied:

```
int -> (int -> bool)
```

Substitutions

solution 1:

b->(int->bool) / a
int->bool / c
b / b

type b -> c with solution applied:

b -> (int -> bool)

solution 2:

int->(int->bool) / a
int->bool / c
int / b

type b -> c with solution applied:

int -> (int -> bool)

Solution 1 is "more general" – there is more flex.

Solution 2 is "more concrete"

We prefer solution 1.

Substitutions

solution 1:

b->(int->bool) / a
int->bool / c
b / b

solution 2:

int->(int->bool) / a
int->bool / c
int / b

type b -> c with solution applied:

b -> (int -> bool)

type b -> c with solution applied:

int -> (int -> bool)

Solution 1 is "more general" – there is more flex.

Solution 2 is "more concrete"

We prefer the more general (less concrete) solution 1.

Technically, we prefer T to S if there exists another substitution U and for all types t, S (t) = U (T (t))

Substitutions

solution 1:

b->(int->bool) / a
int->bool / c
b / b

solution 2:

int->(int->bool) / a
int->bool / c
int / b

type b -> c with solution applied:

b -> (int -> bool)

type b -> c with solution applied:

int -> (int -> bool)

There is always a *best* solution, which we can a *principal solution*.

The best solution is (at least as) preferred as any other solution.

Examples

Example 1

- q = {a=int, b=a}
- principal solution S:
  - S(a) = S(b) = int
  - S(c) = c   (for all c other than a,b)

Example 2

- q = {a=int, b=a, b=bool}
- principal solution S:
  - does not exist (there is no solution to q)

Unification

- Unification: An algorithm that provides the principal solution to a set of constraints (if one exists)
  - Unification systematically simplifies a set of constraints, yielding a substitution
    - Starting state of unification process: ({ },q)
    - Final state of unification process: (S, { })

solution S	constraints q	solution S	constraints q
{ }	b = b′ list a = a b = b′ list a = b′ -> a′ c = c′ list a′ = c′ d list = c′ list d list = c c′ list = c	[b' -> c'/a] [b' list/b] [c' list/c]	{ }

Unification

- Unification simplifies equations step-by-step until
  - there are no equations left to simplify, or
  - we find basic equations are inconsistent and we fail

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | { } -> S
    | {bool=bool} U q' -> unify q'
    | {int = int} U q' -> unify q'
```

Unification

- Unification simplifies equations step-by-step until
  - there are no equations left to simplify, or
  - we find basic equations are inconsistent and we fail

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | { } -> S
    | {bool=bool} U q’ -> unify q’
    | {int = int} U q’ -> unify q’
    | {a = a} U q’ -> unify q’
```

Unification

- Unification simplifies equations step-by-step until
  - there are no equations left to simplify, or
  - we find basic equations are inconsistent and we fail

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {A -> B = C -> D} U q' ->
       unify S ({A = C, B = D} U q')
```

Unification

- Unification simplifies equations step-by-step until
  - there are no equations left to simplify, or
  - we find basic equations are inconsistent and we fail

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {A -> B = C -> D} U q' ->
       unify S ({A = C, B = D} U q')
```

Unification

- Unification simplifies equations step-by-step until
  - there are no equations left to simplify, or
  - we find basic equations are inconsistent and we fail

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {s=a} U q'
    | {a=s} U q' ->
       unify ([s/a] ∪ S) q'
```

??

Unification

Ideal solution :

int / p
int / q
int / r

unify { } $\{(p \to p \to q = q \to r \to int)\}$

unify { } $\{(p = q), (p = r), (q = int)\}$

unify $\{[q/p]\}$ $\{(p = r), (q = int)\}$

unify $\{[q/p], [r/p]\}$ $\{(q = int)\}$

unify $\{[q/p], [r/p], [int/q]\}$ { }

Unification

- Unification simplifies equations step-by-step until
  - there are no equations left to simplify, or
  - we find basic equations are inconsistent and we fail

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
   match q with
   | …
   | {s=a} U q’
   | {a=s} U q’ ->
     unify ([s/a] ∪ S) q’
```

Unification

- Unification simplifies equations step-by-step until
  - there are no equations left to simplify, or
  - we find basic equations are inconsistent and we fail

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {s=a} U q'
    | {a=s} U q' ->
      unify ([s/a] ∪ [s/a]S) [s/a]q'
```

Any occurrence of type variable a in types within S and q must be replaced by s

Unification

let rec unify S q = match q with
| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

unify {} $\{(p \rightarrow p \rightarrow q = q \rightarrow r \rightarrow int)\}$

unify {} $\{(p = q), (p = r), (q = int)\}$

Unification

let rec unify S q = match q with
| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

unify {} $\{(p \to p \to q \;=\; q \to r \to int)\}$

unify {} $\{(p = q), (p = r), (q = int)\}$

unify $\{[q/p]\}$ $\{(q = r), (q = int)\}$

Unification

let rec unify S q = match q with

| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

unify {} $\{(p \to p \to q \ = \ q \to r \to int)\}$

unify {} $\{(p = q), (p = r), (q = int)\}$

unify $\{[q/p]\}$ $\{(q = r), (q = int)\}$

Unification

let rec unify S q = match q with
| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

unify { } $\{(p \to p \to q = q \to r \to int)\}$

unify { } $\{(p = q), (p = r), (q = int)\}$

unify $\{[q/p]\}$ $\{(q = r), (q = int)\}$

unify $\{[r/p], [r/q]\}$ $\{(r = int)\}$

Unification

let rec unify S q = match q with
| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

unify {} $\{(p \to p \to q = q \to r \to int)\}$

unify {} $\{(p = q), (p = r), (q = int)\}$

unify $\{[q/p]\}$ $\{(q = r), (q = int)\}$

unify $\{[r/p], [r/q]\}$ $\{(r = int)\}$

Unification

let rec unify S q = match q with
| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

unify {} $\{(p \to p \to q = q \to r \to int)\}$

unify {} $\{(p = q), (p = r), (q = int)\}$

unify $\{[q/p]\}$ $\{(q = r), (q = int)\}$

unify $\{[r/p], [r/q]\}$ $\{(r = int)\}$

unify $\{[int/p], [int/p], [int/r]\}$ {}

Unification

- Unification simplifies equations step-by-step until
  - there are no equations left to simplify, or
  - we find basic equations are inconsistent and we fail

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {s=a} U q' ->
    | {a=s} U q' ->
       unify ([s/a] ∪ [s/a]S) [s/a]q'
```

Occurs Check

```
G ⊢ u1 ==> e1 : t1, q1
G ⊢ u2 ==> e2 : t2, q2            (for fresh a)
-------------------------------------------------
G ⊢ u1 u2==> e1 e2     :     a,    q1 U q2 U {t1 = t2 -> a}
```

- Consider a program:
  - fun x -> x x

```
# fun x -> x x ;;
Line 1, characters 11-12:
Error: This expression has type 'a -> 'b but an expression was expected of type 'a
The type variable 'a occurs inside 'a -> 'b
```

  - fun (x:'a) -> ((x x):'b)

Occurs Check

G ⊢ u1 ==> e1 : t1, q1
G ⊢ u2 ==> e2 : t2, q2 (for fresh a)

G ⊢ u1 u2==> e1 e2 : a, q1 U q2 U {t1 = t2 -> a}

- Consider a program:
  - fun x -> x x

```
# fun x -> x x ;;
Line 1, characters 11-12:
Error: This expression has type 'a -> 'b but an expression was expected of type 'a
The type variable 'a occurs inside 'a -> 'b
```

  - fun (x:’a) -> ((x x):’b)

- It generates the constraints: ’a = ’a > ’b
- What is the solution to {’a = ’a > ’b}?

Occurs Check

- Consider a program:
  - fun x -> x x
- It generates the constraints: ’a = ’a > ’b
- What is the solution to {’a = ’a > ’b}?
- There is none!

For a constraint {a = s}, whenever a appears in TypeVars(s) and s is not just a, there is no solution to the system of constraints.

Occurs Check

- Consider a program:
  - fun x -> x x
- It generates the constraints: ’a = ’a > ’b
- What is the solution to {’a = ’a > ’b}?

- There is none!

“when a is not in TypeVars(s)” is known as the “occurs check”

Unification

- Unification simplifies equations step-by-step until
  - there are no equations left to simplify, or
  - we find basic equations are inconsistent and we fail

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {a=s} U q' ->
       unify ([s/a] ∪ [s/a]S) [s/a]q'
       when a is not in TypeVars(s)
```

Summary: Unification Engine

(S, {bool=bool} U q) ➔ (S, q)

(S, {int=int} U q) ➔ (S, q)

(S, {a=a} U q) ➔ (S, q)

(S, {A->B = C->D} U q) ➔ (S, {A = C} U {B = D} U q)

(S, {a=s} U q) ➔ ([s/a] U [s/a]S, [s/a]q) when a is not in TypeVars(s)

Irreducible States

- Recall unification simplifies equations step-by-step untial
  - There are no equations left to simplify
  - Or we find basic equations that are inconsistent:
    - int = bool
    - s1->s2 = int
    - s1->s2 = bool
    - a = s (s is a function type and s contains a)

    or is symmetric to one of the above

In the latter case, the program does not type check.

Polymorphism

The type for map looks like this:

```
map : (‘a -> ‘b) -> ‘a list -> ‘b list
```

This type includes an implicit quantifier at the outermost level. So really, map’s type is this one:

```
map : forall ‘a, ‘b. (‘a -> ‘b) -> ‘a list -> ‘b list
```

To use a value with type forall ‘a, ‘b. t, we first substitute types for parameters ‘a, ‘b.   eg:

```
map  (fun x -> x + 1) [2;3;4]
```

here, we substitute [int/’a][int/’b]
in map’s type and then use map at type
(int -> int) -> int list -> int list

Generalization

- OCaml has universal types on the outside ("prenex quantification")

```
forall 'a,'b. (('a -> 'b) -> 'a list -> 'b list)
```

- It does not have types like this:

```
(forall 'a.'a -> int) -> int -> bool
```

```
Argument type has its
own polymorphic quantifier
```

Generalization

- Consider this program:

```
let f g = (g true, g 3)
```

- Notice that parameter g is used inside f as if:
  - 1. Its argument can have type bool, *And*
  - 2. its argument can have type int
- Does this type work?

```
f:  ('a -> int) -> int * int
```

Generalization

- Consider this program:

```
let f g = (g true, g 3)
```

- Notice that parameter g is used inside f as if:
  - 1. Its argument can have type bool, *And*
  - 2. its argument can have type int
- Does this type work?

```
f:  ('a -> int) -> int * int
```

- No: this type say g's argument can be of any type
- Consider the program: f (fun x -> x + 2)

Generalization

- Consider this program again:

```
let f g = (g true, g 3)
```

- We may want to give it this type:

```
f : (forall a.a->a) -> bool * int
```

- Notice that the universal quantifier appears left of ->

Generalization

- This is System F type system.
- System F is a lot like OCaml, except that it allows universal quantifiers in any position. It could type check f.

```
let f g = (g true, g 3)
```

```
f : (forall a.a->a) -> bool * int
```

Generalization

- This is System F type system.
- System F is a lot like OCaml, except that it allows universal quantifiers in any position. It could type check f.

```
let f g = (g true, g 3)
```

```
f : (forall a.a->a) -> bool * int
```

- Unfortunately, type inference in System F is undecidable.

Generalization

- Where do we introduce polymorphic values? Consider:

```
g (fun x -> 3)
```

- It is tempting to do something like this:

```
(fun x -> 3) : forall a. a -> int
```

```
g : (forall a. a -> int) -> int
```

- But we may run into decidability issues

Generalization

- Where do we introduce polymorphic values?

- In OCaml: only when values bound in "let declarations"

```
g (fun x -> 3)
```

No polymorphism for fun x -> 3!

```
let f : forall a. a -> a = fun x -> 3 in
g f
```

Yes polymorphism for f!

Let Polymorphism

- Where do we introduce polymorphic values?

```
let x = v
```

- Rule:
  - If v is a value
  - and v has type scheme s
  - and s has type variables a, b, c, …
  - and a, b, c, … do not appear in the type of other values in the context
  - Then x can have type forall a, b, c. s

Unsound Generalization Example

- Consider this function f - a fancy identity function:

```
let f = fun x -> let y = x in y
```

- A sensible type for f would be:

```
f : forall a. a -> a
```

Unsound Generalization Example

- Consider this function f - a fancy identity function:

```
let f = fun x -> let y = x in y
```

- A bad (unsound) type for f would be:

```
f : forall a, b. a -> b
```

Unsound Generalization Example

- Consider this function f - a fancy identity function:

```
let f = fun x -> let y = x in y
```

- A bad (unsound) type for f would be:

```
f : forall a, b. a -> b
```

```
(f true) + 7
```

goes wrong! but if f can have the bad type,
it all type checks. This *counterexample* to soundness shows
that f can't possible be given the bad type safely

Unsound Generalization Example

- Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

then we
can use y
as if it has
any type,
such as y : b

x : a

suppose we generalize and allow y : forall a.a

but now we have inferred that (fun x -> ...) : a -> b
and if we generalize again,
f : forall a,b. a -> b

That's the bad type!

Unsound Generalization Example

- Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

x : a

suppose we generalize and allow y : forall a.a

this was the bad step – y can't really have any type at all. Its type has got to be the same as whatever the argument x is.

x was in the context when we tried to generalize y!

Let Generalization *Only!*

- In OCaml: only when values bound in “let declarations”

```
g (fun x -> 3)
```

No polymorphism for fun x -> 3!

```
let f : forall a. a -> a = fun x -> 3 in
g f
```

Yes polymorphism for f!

```
# (fun x -> x) (fun x -> 3);;
- : '_weak1 -> int = <fun>
```

A “weak” type variable cannot be generalized. Means “I don’t know what type this is but it can only be one particular type”

Let Generalization *Only!*

- In OCaml: only when values bound in “let declarations”

```
g (fun x -> 3)
```

No polymorphism for fun x -> 3!

```
let f : forall a. a -> a = fun x -> 3 in
g f
```

Yes polymorphism for f!

```
# let g = (fun x -> x) (fun x -> 3);;
val g : '_weak2 -> int = <fun>
# let a = g 1 in g true;;
Error: This expression has type bool but an expression was expected of type
         int
```

Let Generalization *Only!*

- In OCaml: only when values bound in "let declarations"

g (fun x -> 3)

No polymorphism for fun x -> 3!

let f : forall a. a -> a = fun x -> 3 in
g f

Yes polymorphism for f!

```
# let g = fun x -> 3;;
val g : 'a -> int = <fun>
# let _ = g 1 in g true;;
- : int = 3
```

Polymorphic Type Inference Algorithm

```
t ::= a | int | t->t

σ ::= t | ∀a.σ

e ::=
   x
   | n
   | let x = e in e
   | fun x -> e
   | e e
```

- Types
- Type Schemes
- Expressions

Polymorphic Type Inference Algorithm

$$\sigma = \forall \texttt{a1,…,an.t}$$

- Type scheme σ can be instantiated to a type t' by substituting types for the bound variables of σ, i.e.,
  - t' = S σ. For some S s.t. Dom(S) ⊆ BV(σ)
  - t' is said to be an instance of σ (σ > t')
  - t' is said to be a generic instance of σ when S maps variables to new variables.
- Eample:
  - σ = ∀a1. a1 -> a2
  - a3 -> a2 is a generic instance of σ, int -> a2 is not.

Polymorphic Type Inference Algorithm

Gen(G, t) = $\forall a_1, \ldots, a_n$. t where $\{a_1, \ldots, a_n\}$ = TyVar(t) – TyVar(G)

- Generalization introduces polymorphism
- Quantify type variables that are free in t but not free in the type environment G.
- Captures the notion of new type variables of t.

Polymorphic Type Inference Algorithm

(App)

G |- e1: t->t'   G |- e2: t

G |- (e1 e2): t'

(Abs)

G; x:t |- e: t'

G |- (fun x:t -> e): t -> t'

(Var)

(x:σ) ∈ G $\sigma \geq$ t

G |- x: t

x can be considered of type t as long as its type as specified in the environment can be specialized to t

(Const)

G |- n:int

x has a different type in e1 than in e2. In e1, x is not a polymorphic type, but in e2 it generalized into one

(Let)

G; x:t |- e1: t G; {x: Gen(G, t)} |- e2: t'

G |- (let x = e1 in e2): t'

Polymorphic Type Inference Algorithm

```
let W(G, e) = match e with
| c -> ({}, int)
| x -> if x ∈ Dom(G) let ∀a₁, …, aₙ. t = G(x) in ({}, [uᵢ/aᵢ]t)
        else Fail
| (fun x -> e) -> let (S1, t1) = W(G; {x:u}, e) in
                  (S1, S1(u) -> t1)
| (e1 e2) -> let (S1, t1) = W(G, e1) in
             let (S2, t2) = W(S1(G), e2) in
             let S3 = unify(S2(t1), t2 -> u) in
             (S3 U S2 U S1, S3(u))
| let x = e1 in e2 ->
             let (S1, t1) = W(G; {x:u}, e1) in
             let S2 = unify(S1(u), t1) in
             let σ = Gen(S2S1(G), S2(t1)) in
             let (S3, t2) = W(S2S1(G); {x:σ}, e2) in
             in (S3 U S2 U S1, t2)
```

u's represent new type variables

Polymorphic Type Inference Algorithm

```
λx. let f = λy.x          B     A
    in (f 1, f True)
```

W(∅, A) =

  W({x:u1}, B) =

   W({x:u1,f:u2}, (fun y -> x)) = ([], u3->u1)

    W({x:u1,f:u2,y:u3}, x) = ([], u1)

Polymorphic Type Inference Algorithm

```
λx. let f = λy.x      B      A
    in (f 1, f True)
```

W(∅, A) =
  W({x:u1}, B) =
    W({x:u1,f:u2}, (fun y -> x)) = ([], u3->u1)
      W({x:u1,f:u2,y:u3}, x) = ([], u1)
    unify(u2, u3->u1) = [(u3->u1)/u2]
    Gen({x:u1}, u3->u1) = ∀u3. u3-> u1

Polymorphic Type Inference Algorithm

```
λx. let f = λy.x        B   A
    in (f 1, f True)
```

W(∅, A) =
  W({x:u1}, B) =
    W({x:u1,f:u2}, (fun y -> x)) = ([], u3->u1)
      W({x:u1,f:u2,y:u3}, x) = ([], u1)
    unify(u2, u3->u1) = [(u3->u1)/u2]
    Gen({x:u1}, u3->u1) = ∀u3. u3-> u1
    W({x:u1, f:∀u3. u3-> u1}, (f 1)) =
      W({x:u1, f:∀u3. u3-> u1}, f) = ([], u4->u1)

Polymorphic Type Inference Algorithm

```
λx. let f = λy.x      B      A
    in (f 1, f True)
```

W(∅, A) =
  W({x:u1}, B) =
    W({x:u1,f:u2}, (fun y -> x)) = ([], u3->u1)
      W({x:u1,f:u2,y:u3}, x) = ([], u1)
    unify(u2, u3->u1) = [(u3->u1)/u2]
    Gen({x:u1}, u3->u1) = ∀u3. u3-> u1
    W({x:u1, f:∀u3. u3-> u1}, (f 1)) =
      W({x:u1, f:∀u3. u3-> u1}, f) = ([], u4->u1)
      W({x:u1, f:∀u3. u3-> u1}, 1) = ([], int)
      unify(u4->u1, Int->u5) = [int/u4, u1/u5]

Polymorphic Type Inference Algorithm

$$\boxed{\lambda x.\ \boxed{\begin{array}{l} \textit{let } f = \lambda y.x \quad B \\ \textit{in } (f\ 1,\ f\ \mathrm{True}) \end{array}} \quad A}$$

W(∅, A) =
  W({x:u1}, B) =
    W({x:u1,f:u2}, (fun y -> x)) = ([], u3->u1)
      W({x:u1,f:u2,y:u3}, x) = ([], u1)
    unify(u2, u3->u1) = [(u3->u1)/u2]
    Gen({x:u1}, u3->u1) = ∀u3. u3-> u1
    W({x:u1, f:∀u3. u3-> u1}, (f 1)) = (…, u1)
      W({x:u1, f:∀u3. u3-> u1}, f) = ([], u4->u1)
      W({x:u1, f:∀u3. u3-> u1}, 1) = ([], int)
      unify(u4->u1, Int->u5) = [int/u4, u1/u5]
     W({x:u1, f:∀u3. u3-> u1}, (f true)) = (…, u1)

Polymorphic Type Inference Algorithm

```
λx. let f = λy.x            B    A
    in (f 1, f True)
```

W(Ø, A) = (..., (u1 -> (u1,u1)))

W({x:u1}, B) = (..., (u1,u1))

W({x:u1,f:u2}, (fun y -> x)) = ([], u3->u1)

W({x:u1,f:u2,y:u3}, x) = ([], u1)

unify(u2, u3->u1) = [(u3->u1)/u2]

Gen({x:u1}, u3->u1) = ∀u3. u3-> u1

W({x:u1, f:∀u3. u3-> u1}, (f 1)) = (..., u1)

W({x:u1, f:∀u3. u3-> u1}, f) = ([], u4->u1)

W({x:u1, f:∀u3. u3-> u1}, 1) = ([], int)

unify(u4->u1, Int->u5) = [int/u4, u1/u5]

W({x:u1, f:∀u3. u3-> u1}, (f true)) = (..., u1)

Polymorphic Type Inference Algorithm

- It is sound with respect to the type system.
  - An inferred type is verifiable.
- It generates most general types of expressions.
  - Any verifiable type is inferred.
- Complexity
  - PSPACE-Hard