Lecture 12: Type inference

David Bremner

February 19, 2025

Examples from plait

Types are inferred

infer

```
(define f
  (lambda (x y)
    (if x
        (+ y 1) (+ y 2))))
```

Lack of consistency is inferred

noinfer

```
(lambda (x)
  (if x
      (+ x 1) (+ x 2)))
```

A plan for inference

- recursively visit each sub-expression, generating "constraints"
- "solve" those constraints

Type from use

- Consider

```
(lambda (x : ?) (+ x 1))
```

- $\mathtt{x}$ is only used in +
- We have the following rule

$$\frac{\vdash \mathtt{e1} : \mathrm{Num} \qquad \vdash \mathtt{e2} : \mathrm{Num}}{\vdash \mathtt{(+\ e1\ e2)} : \mathrm{Num}}$$

- So $\mathtt{x}$ must have type Num

Unique name

- It will be convenient to assume that each variable has a unique name
- So convert

```
(let ([x 3])
  (+ (let ([x 4])
       x)
     x))
```

stacker

- into

```
(let ([x 3])
  (+ (let ([y 4])
       y)
     x))
```

stacker

Unique name

- It will be convenient to assume that each variable has a unique name
- So convert

```
(let ([x 3])
  (+ (let ([x 4])
       x)
     x))
```

- into

```
(let ([x 3])
  (+ (let ([y 4])
       y)
     x))
```

1. As the book notes this kind of renaming is called α-conversion
2. This is mainly for the discussion; an actual inference algorithm would typically used some kind of scoped environment just like an evaluator or a type calculator, so there is no ambiguity which variable a particular identifier refers to

Type from use II

```
(lambda (x y)
  (if x
      (+ y 1)
      (+ y 2)))
```

- From the (unique) rule for `if`, we learn $\vdash \texttt{x} : \mathrm{Bool}$
- From the (unique) rule for `+`, we learn $\vdash \texttt{y} : \mathrm{Num}$

(lack of) Type from use

```
(lambda (x)
  (if x
      (+ x 1)
      (+ x 2)))
```

- From the (unique) rule for `if`, we learn $\vdash \texttt{x} : \mathrm{Bool}$
- From the (unique) rule for +, we learn $\vdash \texttt{x} : \mathrm{Num}$
  - at this point we detect a contradiction

Many possible types

```
(lambda (x y)
  (if x y y))
```

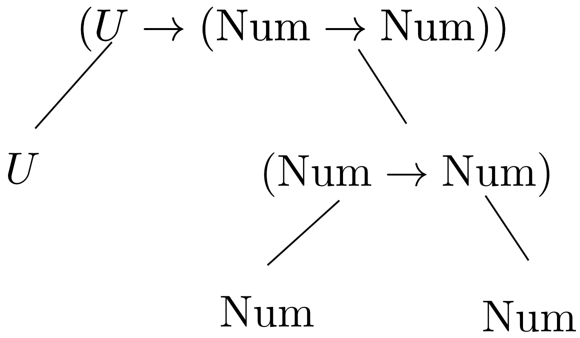
- as before we learn $\vdash \mathtt{x} : \mathrm{Bool}$
- The use of $\mathtt{y}$ doesn't narrow down the type, so we report something like $(\mathrm{Bool}\ T \to T)$

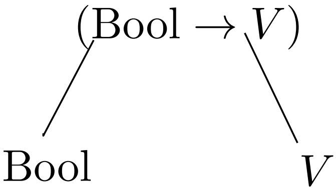
Inference via unification

ti

```
(define (typecheck [exp : Exp] [env : TypeEnv]) : Type
  (type-case Exp exp
    ...
    [(plusE l r)
     (begin
       (unify! (typecheck l env) (numT) l)
       (unify! (typecheck r env) (numT) r)
       (numT))]
    ...))
```

Unification example





Unification algorithm I/II

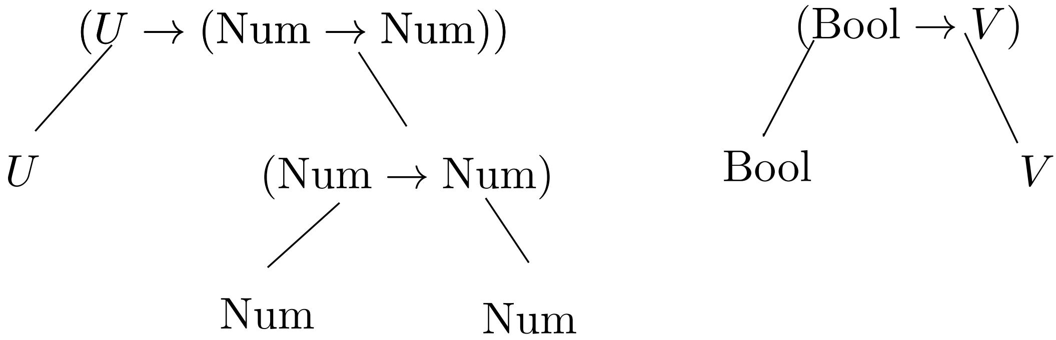
Unify a type τ_1 to type τ_2:

p. 147

ti

- If τ_1 (τ_2) is a type variable T, then unify T and τ_1 (τ_2).
- If τ_1 and τ_2 are both num or bool, succeed
- If τ_1 is $(\tau_3 \to \tau_4)$ and τ_2 is $(\tau_5 \to \tau_6)$, then
  - unify τ_3 with τ_5
  - unify τ_4 with τ_6
- Otherwise, fail

Unification example



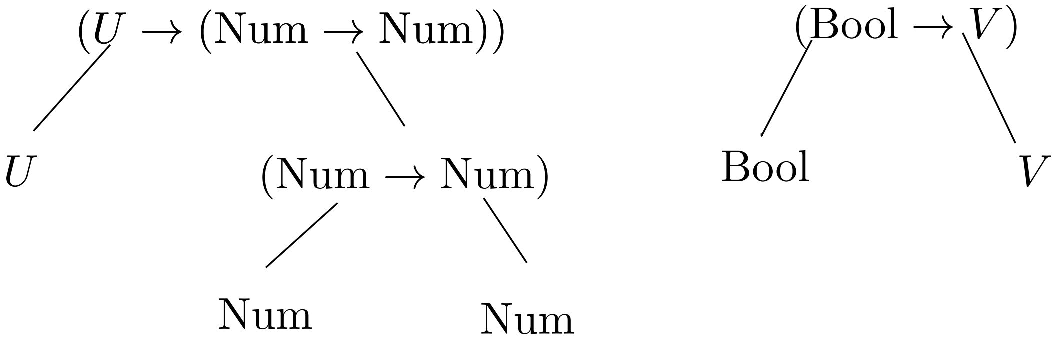
Unification algorithm II/II

Unify a type variable T with a type τ_2:

ti

- If T is set to τ_1, unify τ_1 and τ_2
- If τ_2 is already equivalent to T, succeed
- If τ_2 contains T, then fail
- Otherwise, set T to τ_2 and succeed

Unification example



Implementing type variables

tvar

```
(define-type Type
  [numT]
  [boolT]
  [arrowT (arg : Type) (result : Type)]
  [varT (id : Number) (val : (Boxof (Optionof Type)))])

(define the-box (box (none)))
(define tau1 (arrowT (varT (gen-tvar-id!) the-box)
   (numT)))
(define tau2 (arrowT (varT (gen-tvar-id!) the-box)
   (numT)))
tau1 tau2
(set-box! the-box (some (boolT))) tau1
```

Type inferring function application

ti

```
[(appE fn arg)
 (let ([r-type (varT (gen-tvar-id!) (box (none)))]
       [a-type (typecheck arg env)]
       [fn-type (typecheck fn env)])
   (begin
     (unify! (arrowT a-type r-type) fn-type fn)
     r-type))]
```