12. Types: inference, variants, and unions

David Bremner

February 21, 2024

Examples from plait

Types are inferred

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

Lack of consistency is inferred

```
(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))
```

- x is only used in +
- We have the following rule

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

- So x must have type Num

Unique name

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

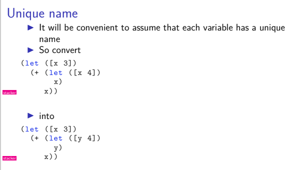
```
(let ([x 3])
  (+ (let ([x 4])
       x)
     x))
```

stacker

- into

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

stacker



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

Inference via unification

```
(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 algorithm I/II

Unify a type τ_1 to type τ_2:

p. 147

- If τ_2 is a type variable T, then unify T and τ_1
- If τ_1 and τ_2 are both num or bool, succeed
- If τ_1 is $(\tau_3 \to\ 4)$ and τ_2 is $(\tau_5 \tau_6)$, then
    - unify τ_3 with τ_5
    - unify τ_4 with τ_6
- Otherwise, fail

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

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))]
```

define-type

p. 150

bt1

```
(define-type BT
  [mt]
  [node (v : Number) (l : BT) (r : BT)])
```

- Binds the name BT in the current type environment
- Supports recursion
- Defines two variants `mt` and `node`
- What are the types of `mt` and `node`?

Generated bindings

Constructors

```
(mt : ( -> BT))
(node : (Number BT BT -> BT))
```

Predicates

```
(mt? : (BT -> Boolean))
(node? : (BT -> Boolean))
```

Accessors

```
(node-v : (BT -> Number))
(node-l : (BT -> BT))
(node-r : (BT -> BT))
```

The problematic ones

p. 151

```
(node-v : (BT -> Number))
(node-l : (BT -> BT))
(node-r : (BT -> BT))
```

bt2

```
(define (size-wrong (t : BT))
  (+ 1 (+ (size-wrong (node-l t))
          (size-wrong (node-r t)))))
(size-wrong (mt))
```

Safely accessing variants

p. 152

bt3

```
(define (size-pm t)
  (type-case BT t
    [(mt) 0]
    [(node v l r) (+ 1 (+ (size-pm l) (size-pm r)))]))
(size-pm (mt))
```

Typechecking type-case

Each `define-type` *extends* the type checker.

p. 153

$$\frac{\Gamma \vdash \texttt{e} : BT \quad \Gamma \vdash \texttt{e1} : T \quad \Gamma[V \leftarrow \mathrm{Num}, L \leftarrow \mathrm{BT}, R \leftarrow \mathrm{BT}] \vdash \texttt{e2} : T}{\Gamma \vdash \texttt{(type-case BT e [(mt) e1][(node V L R) e2])} : T}$$

- can be automatically generated

Space usage for algebraic data type

p. 153

- Can desugar pattern matching into cond
- Need some level of "local" tags for predicates

bt4

```
(define (size-pm-ds (t : BT))
  (cond
    [(mt? t) 0]
    [(node? t)
     (let ([v (node-v t)]
           [l (node-l t)]
           [r (node-r t)])
       (+ 1 (+ (size-pm-ds l) (size-pm-ds r))))]))
```

Retrofitted type systems

- A common (recent) strategy is to add a static type system to existing dynamically typed languages
- Examples include `Typescript` and `mypy` for python.
- The general idea is convert <u>some</u> run-time errors into compile-time errors.

Another approach to variants

Variant information is lost to the type system. p. 155

```
(node : (Number BT BT -> BT))
```

Union types

In "typed/racket" we have another option.

bt5

```
(define-type-alias BT (U mt node))
(struct mt ())
(struct node ([v : Number] [l : BT] [r : BT]))
```

- what constructors, predicates, and accessors are defined?

Using our union type

bt6

```
(define t1
  (node 5
        (node 3
              (node 1 (mt) (mt))
              (mt))
        (node 7
              (mt)
              (node 9 (mt) (mt)))))
```

Computing the size

- Does the following typecheck? Why or why not?

bt7

```
(define (size-tr [t : BT]) : Number
  (cond
    [(mt? t) 0]
    [(node? t) (+ 1 (size-tr (node-l t)) (size-tr
       (node-r t)))]))
```

Predicates are special in typed/racket

- This program has a bug. Does the type checker catch it or not?

bt8

```
(define (size-tr [t : BT]) : Number
  (cond
    [(node? t) 0]
    [(mt? t) (+ 1 (size-tr (node-l t)) (size-tr
       (node-r t)))]))
```

Predicates are special in typed/racket

- This program has a bug. Does the type checker catch it or not?

```
(define (size-tr [t : BT]) : Number
  (cond
    [(node? t) 0]
    [(mt? t) (+ 1 (size-tr (node-l t)) (size-tr
       (node-r t)))]))
```

1. The type errors from typed/racket are much better that those from plait, at least here