Assignment 11: Type Inference

> It is reasonable to hope that the relationship between computation and logic in the next century will be as fruitful as that between calculus and physics in the last.
>
> —*John McCarthy, 1967*

Assignment

Part I

You should download, print, and complete a11-partI. This has questions from the reading, and asks you to infer by hand the types of expressions as we did in class on Tuesday. **Bring your solution to class on Thursday!**

Part II

The rest of your assignment this week is to complete your own type inferencer and inhabiter in miniKanren. **You should start with** a11.rkt, the inferencer we began on Thursday, and add to it `*`, `not`, `zero?`, `sub1`, and `fix`. To be clear, your solution should not involve `pmatch` at all; it should be written entirely in miniKanren. You can use the a11-student-tests.rkt file to test your file.

You may find the following "old-school" notes from Dan, Will, and David of some use.

Your extended inferencer should pass all of the test cases provided for the regular portion of the assignment. Honors students and real go-getters should adjust the expectations accordingly.

To avoid divergence for some of the tests, you must carefully order goals within a conjunction to “fail fast”. Keep these rules in mind:

- Simple unifications should come before recursive calls. For example, `(== 5 x)` should come before `(foo y z)`.
- Recursive calls with instantiated arguments should come before recursive calls with uninstantiated arguments.

For example, `(foo `(,x ,y ,z) 5)` should come before `(foo u v)`, assuming `x`, `y`, `z`, `u`, and `v` are fresh.

You might have seen Dan and Jason, in class, add a relation `is-a-typeo` that ensured in the printed final answer, no type miniKanren returns has any fresh variables in it. Don't do this, as it crowds out interesting answers with uninteresting ones. If you don't recall this, don't worry about it.

```
> (run* (q) (!- '() #t q))
(Bool)
> (run* (q) (!- '() 17 q))
(Nat)
> (run* (q) (!- '() '(zero? 24) q))
(Bool)
> (run* (q) (!- '() '(zero? (sub1 24)) q))
(Bool)
> (run* (q) (!- '() '(not (zero? (sub1 24))) q))
(Bool)
> (run* (q)
    (!- '() '(zero? (sub1 (sub1 18))) q))
(Bool)
> (run* (q)
    (!- '()  '(lambda (n) (if (zero? n) n n)) q))
((Nat -> Nat))
> (run* (q)
    (!- '() '((lambda (n) (zero? n)) 5) q))
(Bool)
> (run* (q)
    (!- '() '(if (zero? 24) 3 4) q))
(Nat)
> (run* (q)
    (!- '() '(if (zero? 24) (zero? 3) (zero? 4)) q))
(Bool)
> (run* (q)
    (!- '() '(lambda (x) (sub1 x)) q))
((Nat -> Nat))
> (run* (q)
    (!- '() '(lambda (a) (lambda (x) (+ a x))) q))
((Nat -> (Nat -> Nat)))
> (run* (q)
    (!- '() '(lambda (f)
```

```
               (lambda (x)
                 ((f x) x)))
         q))
(((_.0 -> (_.0 -> _.1)) -> (_.0 -> _.1)))
> (run* (q)
    (!- '() '(sub1 (sub1 (sub1 6))) q))
(Nat)
> (run 1 (q)
    (fresh (t)
      (!- '() '(lambda (f) (f f)) t)))
()
> (length (run 20 (q)
             (fresh (lam a b)
               (!- '() `((,lam (,a) ,b) 5) 'Nat)
               (== `(,lam (,a) ,b) q))))
20
> (length (run 30 (q) (!- '() q 'Nat)))
30
> (length (run 30 (q) (!- '() q '(Nat -> Nat))))
30
> (length (run 500 (q) (!- '() q '(Nat -> Nat))))
500
;; At this point, stop and take a look at maybe the 500th
;; program you generate
;; (last (run 500 (q) (!- '() q '(Nat -> Nat))))
;; You should be amazed at how quickly it's generating them.
;; If it isn't fast, consider reordering your clauses.
> (length (run 30 (q) (!- '() q '(Bool -> Nat))))
30
> (length (run 30 (q) (!- '() q '(Nat -> (Nat -> Nat)))))
30
> (length (run 100 (q)
             (fresh (e t)
               (!- '() e t)
               (== `(,e ,t) q))))
100
> (length (run 100 (q)
             (fresh (g e t)
               (!- g e t)
               (== `(,g ,e ,t) q))))
100
> (length
   (run 100 (q)
     (fresh (g v)
       (!- g `(var ,v) 'Nat)
       (== `(,g ,v) q))))
100
> (run 1 (q)
       (fresh (g)
  (!- g
       '((fix (lambda (!)
         (lambda (n)
    (if (zero? n)
        1
        (* n (! (sub1 n)))))))
  5)
       q)))
(Nat)
> (run 1 (q)
       (fresh (g)
  (!- g
       '((fix (lambda (!)
         (lambda (n)
    (* n (! (sub1 n))))))
  5)
       q)))
(Nat)
>
```

Brainteaser

Extend your type inferencer to recognize *pair types*. You will need to add support for a `pairof` type to the type inferencer such that the following tests pass.

```
> (run* (q) (!- '() '(cons (zero? 1) (zero? 0)) q))
((pairof Bool Bool))
> (run* (q) (!- '() '(cons (zero? 1) (cons (zero? 1) (zero? 0))) q))
((pairof Bool (pairof Bool Bool)))
> (run* (t) (!- '() '(lambda (x) (cons x x)) t))
(( .0 -> (pairof  .0  .0)))
```

```
> (run* (t) (!- '() '(lambda (x) (lambda (y) (cons (zero? x) (+ x y)))) t))
((Nat -> (Nat -> (pairof Bool Nat))))
```

Once you've done the above, add two more clauses to the type inferencer such that the following tests pass. (By the way, these two new lines will be quite similar to each other!)

```
;; a function that accepts a pair of an Nat and anything
> (run* (t) (!- '() '(lambda (x) (zero? (car x))) t))
(((pairof Nat _.0) -> Bool))
> (run* (t) (!- '() '((lambda (x) (zero? (car x))) (cons 0 1)) t))
(Bool)
> (run* (t) (!- '() '((lambda (x) (zero? (car x))) (cons 0 #f)) t))
(Bool)
> (run* (t) (!- '() '((lambda (x) (zero? (car x))) (cons #f 0)) t))
()
;; a function that accepts a pair of anything and an Nat
> (run* (t) (!- '() '(lambda (x) (zero? (cdr x))) t))
(((pairof _.0 Nat) -> Bool))
> (run* (t) (!- '() '((lambda (x) (zero? (cdr x))) (cons 0 1)) t))
(Bool)
> (run* (t) (!- '() '((lambda (x) (zero? (cdr x))) (cons 0 #f)) t))
()
> (run* (t) (!- '() '((lambda (x) (zero? (cdr x))) (cons #f 0)) t))
(Bool)
```

Just Dessert: Let Polymorphism

Extend the type inferencer so that the following type is inferred:

```
> (run* (q)
    (!-o '() '(let ([f (lambda (x) x)])
                        (if (f #t) (f (cons (f 4) 5)) (f (cons 5 (f 6))))))
        q))
((pairof Nat Nat))
```

lp-a1.txt · Last modified: 2014/11/14 02:56 by jhemann