Problem Set 8: Types

Problem 1

We can define the Scheme syntax of the simply typed lambda calculus with one built-in type Int using the following BNF specification:

\[
<\text{Exp}> ::= (\text{<Identifier>} \text{<Type>}) \\
\quad | (\text{lambda} (\text{<Identifier>} \text{<Type>}) \text{<Exp>} \text{<Type>}) \\
\quad | ((\text{<Exp>} \text{<Exp>}) \text{<Type>}) \\
\]

\[
<\text{Type}> ::= \text{Int} \\
\quad | (\text{<Type> -< Type>}) \\
\]

The meta-symbol \text{<Identifier>} stands for a Scheme symbol that is different from the symbol \text{lambda}.

We can think of every typed lambda calculus term as a pair of a value (of the untyped lambda calculus) and a tag to indicate its type, written \((\text{exp type})\).

- Using the \text{define-datatype} abstraction, define the abstract syntax of data types \text{Exp} and \text{Type} according to the BNF specification given above. Use as variant names the name given in the boxes.
- Define the procedure \text{parse-exp lst} that when applied to a list representation of a typed lambda calculus expression will return its abstract syntax tree. Apply the principle “Follow the Grammar”. That is, you need to define a procedure for every syntactic category.
- Given a typed lambda calculus expression \(e \in <\text{Exp}>\) define the predicate \text{type-annotations-consistent?}, which when applied to \(e\) returns \#t if all
type annotations in e are consistent. Otherwise, the predicate returns \#f. You have to implement a case analysis using the cases statement:

\[\begin{align*}
\text{o e == (v t): } & \#t \\
\text{o e == ((lambda (x t_1) b) t_2): } \\
& \text{t_2 is a function type } \&\& \\
& \text{t_1 == argument type of t_2 } \&\& \\
& (\text{type-annotations-consistent? b} ) \&\& \\
& \text{type of b == result type of t_2} \\
\text{o e == ((ftn arg) t): } \\
& (\text{type-annotations-consistent? ftn}) \&\& \\
& (\text{type-annotations-consistent? arg}) \&\& \\
& \text{type of ftn is a function type } \&\& \\
& \text{argument type of type of ftn == type of arg } \&\& \\
& \text{t == result type of type of ftn}
\end{align*}\]

Examples:

\[
> \text{(define var1 '((y ((Int -> (Int -> Int)) -> (Int -> Int))) )}
\]

\[
> \text{(define abs1 '((lambda (x (Int -> Int)) (x (Int -> Int))) } \\
& ((Int -> Int) -> (Int -> Int))) )
\]

\[
> \text{(define appl1 '(((lambda (x (Int -> Int)) (x (Int -> Int))) } \\
& ((Int -> Int) -> (Int -> Int))) } \\
& (y (Int -> Int))) \\
& (Int -> Int) )
\]

\[
> \text{(define applEl '(((x (Int -> Int)) (y (Int -> Int))) Int) )}
\]

\[
> \text{(parse-exp var1)}
\]

\[
(t-id y \\
(t-ftn \\
(t-ftn (t-int) (t-ftn (t-int) (t-int))) \\
(t-ftn (t-int) (t-int))))
\]

\[
> \text{(parse-exp abs1)}
\]

\[
(t-abs \\
\quad x \\
(t-ftn (t-int) (t-int)) \\
(t-id x (t-ftn (t-int) (t-int))) \\
(t-ftn (t-ftn (t-int) (t-int)) (t-ftn (t-int) (t-int))))
\]

\[
> \text{(parse-exp appl1)}
\]

\[
(t-app \\
\quad x
\]
(t-ftn (t-int) (t-int))
(t-id x (t-ftn (t-int) (t-int)))
(t-ftn (t-ftn (t-int) (t-int)) (t-ftn (t-int) (t-int)))
(t-id y (t-ftn (t-int) (t-int)))
(t-ftn (t-int) (t-int)))

> (parse-exp applE1)
(t-app
  (t-id x (t-ftn (t-int) (t-int)))
  (t-id y (t-ftn (t-int) (t-int)))
  (t-int))

> (type-annotations-consistent? (parse-exp var1))
#t
> (type-annotations-consistent? (parse-exp abs1))
#t
> (type-annotations-consistent? (parse-exp appl1))
#t
> (type-annotations-consistent? (parse-exp applE1))
#f

Solution:

(definedatatype Exp Exp?
  (t-id (id symbol?)
    (type Type?))
  (t-abs (id symbol?)
    (id-type Type?)
    (body Exp?)
    (type Type?))
  (t-app (ftn Exp?)
    (arg Exp?)
    (type Type?)))

(definedatatype Type Type?
  (t-int)
  (t-ftn (op Type?)
    (arg Type?)))

(define parse-exp
  (lambda (e)
    (if (and (list? e)
         (= (length e) 2))
      (let ((exp (car e))
            (type (cadr e)))
        (cond
          ((and (symbol? exp)
                (not (eqv? exp 'lambda)))
           ; build variable
           (t-id exp (parse-type type)))
          ((and (list? exp)
                (= (length exp) 3)) ; lambda abstraction
           (symbol? (car exp)) ; keyword lambda
           (list? (cadr exp)) ; formal argument list
           (= (length (cadr exp)) 2) ; formal argument list
           (symbol? (caadr exp))) ; formal variable
           ...)...)
; build abstraction
(t-abs (caadr exp) ; formal variable
 (parse-type (cadadr exp)) ; formal type
 (parse-exp (caddr exp)) ; body
 (parse-type type))) ; type of abs
((and (list? exp)
 (= (length exp) 2))
 ; build application
 (t-app (parse-exp (car exp))
 (parse-exp (cadr exp))
 (parse-type type))
 (else (Error "Typed expression expected!")))
)
)
(Error "Typed expression expected!")
)
)

(define parse-type
define (lambda (t)
 (cond
 ((and (symbol? t) (eqv? t 'Int))
  (t-int))
 ((and (list? t) (= (length t) 3) (eqv? (cadr t) '->))
  (t-ftn (parse-type (car t)) (parse-type (caddr t))))
 (else (Error "Type expected!"))
)
)

; pl/3

(define type-annotations-consistent?
define (lambda (e)
 (cases Exp e
  (t-id (id type) #t)
  (t-abs (id id-type body type)
   (and (is-ftn-type? type)
    (equal? id-type (get-arg-type type))
    (type-annotations-consistent? body)
    (equal? (get-type body) (get-result-type type))))
  (t-app (ftn arg type)
   (and (type-annotations-consistent? ftn)
    (type-annotations-consistent? arg)
    (is-ftn-type? (get-type ftn))
    (equal? (get-arg-type (get-type ftn)) (get-type arg))
    (equal? (get-result-type (get-type ftn)) type)))
  )
)
)

(define is-ftn-type?
define (lambda (t)
 (cases Type t
  (t-ftn (f a) #t)
  (else #f)
)
)
)
(define get-type
  (lambda (e)
    (cases Exp e
      (t-id (id type) type)
      (t-abs (id id-type body type) type)
      (t-app (ftn arg type) type)
    ))
  )
)

(define get-arg-type
  (lambda (t)
    (cases Type t
      (t-ftn (f a) f)
      (else (Error "argument is not a function type!"))
    ))
  )
)

(define get-result-type
  (lambda (t)
    (cases Type t
      (t-ftn (f a) a)
      (else (Error "argument is not a function type!"))
    )
  ))
)

Total: 20 + 60 + 80 = 160

**Submission deadline:** Thursday, April 26, 2007, 2:10 p.m.
**Submission procedure:** on paper in class.