A Gradual Typing Poem

Sam Tobin-Hochstadt & Robby Finder
The Problem

Write a function that accepts the specification of an infinite regular tree and turn it into a representation of the tree
The Problem

Write a function that accepts the specification of an infinite regular tree and turn it into a representation of the tree

Write a function that accepts a tree and finds its period
An Example
An Example

Implementation
An Example

(a (b b c)
 (c (d d d)
  (e e c)))
An Example

\[(a \ (b \ b \ c))\]
\[(c \ (d \ d \ d))\]
\[(e \ e \ c))\]

Implementation

Spec
(a (b b c)
  (c (d d d)
    (e e c))))
The Problem with the Solution

The Standard Solution

- exposes mutability
- exposes placeholders
- pushes the burden onto the client (the period function)
The Problem with the Solution

data STree = STree String STree STree | Link String

data ITree = ITree String ITree

link :: STree -> ITree
link main = conv main
where conv :: STree -> ITree
    conv (STree str t1 t2) = ITree str (conv t1) (conv t2)
    conv (Link str) = find main str []

find :: STree -> String -> STree -> STree
find (STree str1 t1 t2) str pending
    str2==str = conv (STree str2 t1 t2)
    otherwise = find t1 str (tr:pending)
in conv main end

link main = let
    conv (STree (str,t1,t2)) =
        ITree (str,fn () => conv (t1,fn () => conv t2))
    in conv (STree (t2,t1)) str pending =
        if (str2=st2) then conv (STree str2 t1 t2)
        else find (str t:pending) str2 end

bt :: ITree
bt = left at

ct :: ITree
ct = right at

val exists=List.exists
val toString=Int.toString

val at = link (STree("a",STree("b",Link "b", Link "c"),
STree("c",STree("d",Link "d", Link "e"),
STree("e",Link "e", Link "c")))

val bt = left at
val ct = right at

val answers = [period at, period bt, period ct]
Problem Statement

The Typed Scheme Advantage

An Intro to Typed Scheme

The First Solution

The Second Solution

The Moral
Where We’re Going

Typed Scheme allows a simple implementation of the problem where

- the complexity of the implementation is hidden
- the client has all the advantages of the original code
Where We’re Going

Typed Scheme allows a simple implementation of the problem where

- the complexity of the implementation is hidden
- the client has all the advantages of the original code

All because of gradual typing!
Problem Statement

The Typed Scheme Advantage

An Intro to Typed Scheme

The First Solution

The Second Solution

The Moral
Typed Scheme

#lang typed-scheme

(: x Number)
(define x 1)
Typed Structs

#lang typed-scheme

(define-struct: ImpTree
  ([name : Symbol]
   [left : (U ImpTree Symbol)]
   [right : (U ImpTree Symbol)]))
Occurrence Typing

#lang typed-scheme

(if (ImpTree? t)
    (display (ImpTree-name t))
    (display "no name"))
#lang typed-scheme

(: t ImpTree)
(define t (make-ImpTree 'a 'x 'y))
(provide t)
Typed/Untyped Integration

#lang scheme

(provide t)
(define t (make-ImpTree ))

contract boundary

#lang typed-scheme

(require/typed "x.ss" [t ImpTree])
(ImpTree-left t)
Problem Statement

The Typed Scheme Advantage

An Intro to Typed Scheme

The First Solution

The Second Solution

The Moral
(define-type-alias SpecTree
  (Rec ST (U Symbol (List Symbol ST ST)))))
(define-type-alias SpecTree
  (Rec ST (U Symbol (List Symbol ST ST))))
(define-struct: ImpTree
  ([name : Symbol]
   [left : (U ImpTree Symbol)]
   [right : (U ImpTree Symbol)]
  #:mutable)
Client Code

(: period (ImpTree -> (U Number #f)))
(define (period it) )
(: period (ImpTree -> (U Number #f)))
(define (period it)
  (: bfs)
  (define (bfs s v)
    (let ([l (ImpTree-left it)]
          [r (ImpTree-right it)])
      (if (and (ImpTree? l) (ImpTree? r))
        (bfs (list (cons l 1) (cons r 1))
             '())
        (error 'fail)))))
(: period (ImpTree -> (U Number #f)))
(define (period it)
  (: bfs)
  (define (bfs s v)
    (let ([l (ImpTree-left it)]
           [r (ImpTree-right it)])
      (if (and (ImpTree? l) (ImpTree? r))
        (bfs (list (cons l 1) (cons r 1))
             '())
        (error 'fail)))))
 (: bfs ((Listof (Pair ImpTree Number))
   (Listof Symbol)
   -> (U Number #f)))
(define (bfs stack visited)
  (match stack
   ['() #f]
   [(cons (cons (struct ImpTree (str2 tl tr)) i) rest)
     (cond
      [(eq? str2 (ImpTree-name it)) i]
      [(memq str2 visited) (bfs rest visited)]
      [(and (ImpTree? tl) (ImpTree? tr))
        (bfs (append rest (list (cons tl (add1 i))
                                 (cons tr (add1 i))))
             (cons str2 visited))]
      [else (error 'fail)])]))
Client Code

Exactly the problem we thought we’d have
Problem Statement

The Typed Scheme Advantage

An Intro to Typed Scheme

The First Solution

The Second Solution

The Moral
Better Specification

```
(define-type-alias SpecTree
  (Rec ST (U Symbol (List Symbol ST ST))))
(define-struct: ImpTree
  ([name : Symbol]
   [left : ImpTree]
   [right : ImpTree]))
```
(require/typed "itree.ss"

[struct ImpTree ([name : Symbol]
    [left : ImpTree]
    [right : ImpTree])]

[link (SpecTree -> ImpTree)])
Client Code

(: period (ImpTree -> (U Number #f)))
(define (period it) )
(: period (ImpTree -> (U Number #f)))
(define (period it)
  (let ([l (ImpTree-left it)]
         [r (ImpTree-right it)])
    (bfs (list (cons l 1) (cons r 1)) '
())))
(: bfs ((Listof (Pair ImpTree Number))
   (Listof Symbol)
   -> (U Number #f)))

(define (bfs stack visited)
  (match stack
    ['() #f]
    [(cons (cons (struct ImpTree (str2 tl tr)) i) rest)
     rest]
  (cond
    [(eq? str2 (ImpTree-name it)) i]
    [(memq str2 visited) (bfs rest visited)]
    [else
     (bfs (append rest
           (list (cons tl (add1 i))
                 (cons tr (add1 i)))
           (cons str2 visited)))])})
What Happened?

Typed Scheme automatically synthesized contracts
Mutation is hidden
Problem Statement

The Typed Scheme Advantage

An Intro to Typed Scheme

The First Solution

The Second Solution

The Moral
Gradual Typing adds expressiveness to typed languages
Thank You