



Mutually Referential Data Definitions

Corky Cartwright
Stephen Wong
Rice University



Announcements and Plan

- Reminder: Homework 2 due Friday at 10 am.
- Plan for today
 - What is a mutually referential (inductive/recursive) data definition and corresponding recursion template
 - Simple and deep examples illustrating the approach.



Sample Mutually Referential Data Definition

```
; Descendant trees [compare to ancestor trees]
; A person is a structure
; * (make-person loc n)
;   where loc is a list-of-person (the children
;   of the person), and n is a symbol.

; A list-of-alpha is either
; * empty, or
; * (cons a loa) where a is an alpha, and loa
;   is a list-of-alpha
```



Why Are Mutual References Used Here?

A **person** is a tree node with a *variable* number of subtrees (children). Hence, the children component of a person must be a list, which has a separate self-referential definition. Note that the definition of **person** refers to **list-of-person** and the definition of **list-of-person** refers back to **person**.



Mutual Templates

- Common terminology: mutually **recursive/inductive** instead of mutually **referential**
- Writing **one** function on any of these types requires writing a set of functions for **all** the mutually recursive types
- *Each reference to a mutually recursive type in a data definition corresponds to a different recursive call to the appropriate function in the corresponding template.*

The template for **list-of-alpha** is **enhanced** when **list-of-alpha** is used in a mutually referential data definition



Descendant Tree Templates

```
; A person is a structure
; * (make-person n lop)
; where n is a symbol (name of the person) and
; lop (children of the person) is a list-of-person.
; We assume that list-of-alpha has already been defined.
(define-struct person (name children))
; Templates
; person-fn: person -> ...
; (define (person-fn ... p ...)
;   (... (person-name p) ...
;         (lop-fn (person-children ... p ...)) ... ))
; lop-fn: ... list-of-person ... -> ?
; (define (lop-fn ... lop ...)
;   (cond [(empty? lop) ...]
;         [else
;          ... (person-fn ... (first lop) ...) ...
;          ... (lop-fn ... (rest lop) ...) ...]))
```



Templates, cont.

```
; A list-of-person is either
;   empty, or
;   (cons p lop) where
; where p is a person and lop is a list-of-person
; lop-fn: ... list-of-person ... -> ?
; (define (lop-fn ... lop ...)
;   (cond [(empty? lop) ...]
;         [else
;          ... (person-fn ... (first lop) ...) ...
;          ... (lop-fn ... (rest lop) ...) ...) ]))
```

Note: you do not have to rewrite the data definition for (one definition of **list-of-alpha** per file is sufficient, but you do need to rewrite the template for **list-of-alpha** in the context of a mutually recursive definition of **person**)



Function calls in templates

Mutually recursive calls are part of template

- Use of a mutually recursive type is just the same as a recursive use of a type itself
- A set of mutually recursive type definitions is really one big recursive type definition with multiple parts and each part has a template

The form of the function calls in the template(s) is crucial for ensuring termination



More about termination

- For the inductive (self-referential/recursive) types we saw before today, a recursive functions terminates if
 - it handles the base case(s) cleanly, and
 - it only makes recursive calls on substructures of its primary argument, e.g., the **rest** of a non-empty list
- Mutually recursive (referential) definitions are the same
 - Example: Imagine a type **box** that can contain **bags**, and a type **bag** that can contain **boxes**. Why does the template ensure termination?
 - Any box will be bigger than any bag it contains
 - Similarly for bags.
 - No infinite descending chains of containment.



Sample Program on Descendant Trees

```
; person-count : person -> N
; Purpose: (person-count p) counts the number of people in
  the person (descendant tree) p
; lop-count : list-of-person -> N
; Purpose; (lop-count lop) counts the number of people in
  the list-of-person lop
; Template Instantiation
(define (person-count p)
  ... (lop-count (person-children p)) ... )
(define (lop-count alop)
  (cond [(empty? alop) ... ]
        [(cons? Alop) ... (person-count (first alop))
                           ... (lop-count (rest alop)) ... ]))
```




Sample Program on Descendant Trees

; Code

```
(define (person-count p)
  (add1 (lop-count (person-children p))))
(define (lop-count alop)
  (cond [(empty? alop) 0]
        [(cons? Alop)
         (+ (person-count (first alop))
            (lop-count (rest alop))) ]))
```

Note: the preceding code was produced by copying the template instantiation, erasing the ellipsis dots and inserting the text show in red above.



Another Example of Mutually Recursive Data (Unix File System)

```
; A file is either:
;   a raw-file, or
;   a dir (short for directory)

; A dir is a structure
; (make-dir lonf) where lonf is a list-of-namedFile
(define-struct dir (namedFiles))

; A list-of-namedFile is a form of list-of-alpha.

; A namedFile is a structure
; (make-namedFile name f) where name is a symbol and f
; is a file.
(define-struct namedFile (name file))
```

Note: **dir** could be eliminated since it merely wraps a **list-of-namedFile**, but the struct **dir** distinguishes directories from lists.



Templates

```
; ... file ... -> ?
(define (file-fn ... f ...)
  (cond [(raw-file? f) ...] ; process raw file
        [(dir? f) ...
         ... (dir-fn ... f ...) ... ]))

; ... dir ... - > ?
(define (dir-fn ... d ...)
  ... (lonf-fn ... (dir-namedFile d) ...) ... )

; ... list-of-nameFile ... - > ?
(define (lonf-fn ... lonf ... )
  (cond [(empty? lonf) ... ]
        [(cons? lonf) ...
         ... (namedFile-fn ... (first lonf) ...) ... )
         ... (lonf-fn ... (rest lonf) ...) ... ]))

; ... namedfile ... - > ?
(define (namedFile-fn ... nf ...)
  ... (namedFile-name nf) ...
  ... (file-fn ... (namedFile-file nf) ...) ... )
```



Example function on file system

```
; dir-find?: dir symbol -> boolean
; Purpose: (dir-find? d n) determines if a file named n occurs in dir d
; Template Instantiation
(define (dir-find? d n)
  ... (lonf-find? (dir-namedFiles d) n) ... )
(define (lonf-find? lonf n)    ;; is nameFiles-find? a better name
  (cond [(empty? lonf) ...]
        [(cons? lonf)
         ... (namedFile-find? (first lonf) n)
         ... (lonf-find? (rest lonf) n) ... ]))
(define (namedFile-find? nf n)
  ... (namedFile-name nf) ...
  ... (file-find? (namedFile-file nf) n) ... )
(define (file-find? f n)
  (cond [(rawFile? f) ... ]
        [(dir? f) ... (dir-find? f n) ... ]))
```



Code

```
(define (dir-find? d n) (lonf-find? (dir-namedFiles d) n))

(define (lonf-find? lonf n)
  (cond [(empty? lonf) false]
        [(cons? lonf)
         (or (namedFile-find? (first lonf) n)
             (lonf-find? (rest lonf) n))]))

(define (namedFile-find? nf n)
  (or (equal? (namedFile-name nf) n)
      (file-find? (namedFile-file nf) n)))

(define (file-find? f n)
  (cond [(rawFile? f) false]
        [(dir? f) (dir-find? f n)]))
```



For Next Class

- Attend lab and start on homework
- Read assigned portions of HTDP.