Mutually Referential Data Definitions

Corky Cartwright
Department of Computer Science
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.

A 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-person is either
; * empty, or
; * (cons p lop) where p is a person, and lop
;   is a list-of-person
```

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 ***different*** when list-of-alpha is used in a mutually referential data definition

Descendant Tree Templates

```
; A person is a structure
;   (make-person n loc)
; where n is a symbol (the name of the person) and
; loc is a list-of-person.
(define-struct person (name children))

#|
person-fn: person -> ...
(define (person-fn ... p ...)
  (... (person-name p) ...
       (lop-fn (person-children ... p ...))
   ...))
|#
```

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
; loc-fn: list-of-person -> ...
; (define (lop-fn ... lop ...)
;  (cond [(empty? lop) ...]
;        [else
;          ... (person-fn ... (first lop) ...) ...
;          ... (lo-fn ... (rest lop) ...) ...)]))
```

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) 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.

Code

- Write a function that counts the people in a descendant tree

```
; person-count : person -> N
; lop-count : list-of-person -> N
(define (person-count p)
  (add1 (lop-count (person-children p))))
(define (lop-count alop)
  (cond [(empty? alop) 0]
        [else (+ (person-count (first alop))
                 (lop-count (rest alop)))]))
```

Note: Mutual “defines” should be contiguous

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

Templates

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

; A dir is
;(make-dir lonf) where lonf is a list-of-namedFiles

; dir-fn : dir -> ...
(define (dir-fn ... d ...)
   ... (namedFiles-fn ... (dir-namedFiles d) ...) ... )
```

Templates cont.

```
(define (lonf-fn ... lonf ... )
  (cond [(empty? lonf) ... ]
        [(cons? lonf) ...
         ... (namedFile-fn ... (first lonf) ... ) ... )
         ... (lonf-fn ... (rest lonf) ...) ... ]))

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

Example function on file system

```
; find?: dir symbol -> boolean
; Purpose: (find? d n) determines whether a file with name n
  occurs in directory d.
; Instantiated template

(define (find? d n) ... (nFiles-find? (dir-nFiles d) n) ... )

(define (nFiles-find? lonf n)
  (cond [(empty? lonf) ...]
        [(cons? lonf)
         ... (nFile-find? (first lonf) n)
         ... (nFiles-find? (rest lonf) n) ... ]))

(define (nFile-find? nf n)
  ... (nFile-name nf) ...
  ... (file-find? (nFile-file nf) n) ... )
```

Example function cont.

```
(define (nFiles-find? lonf n)
  (cond [(empty? lonf) ...]
        [(cons? lonf)
         ... (nFile-find? (first lonf) n) ...
         ... (nFiles-find? (rest lonf) n) ... ]))

(define (nFile-find? nf n)
  ... (nFile-name nf) ...
  ... (nFile-find? (nFile-file nf) n) ... )

(define (file-find? f n)
  (cond [(rawFile? f) ... ]
        [(dir? f) ... (find? f n) ... ]))
|#
```

Code

```
(define (find? d n) (nFiles-find? (dir-nFiles d) n))

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

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

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

For Next Class

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