
Module 6b: Mutual Reference

CPSC 110

Peyton Seigo

2018-10-15

Module 6b: Mutual Reference

Learning goals

Learn how to use multiple mutually referential types.

- Be able to identify problem domain information of arbitrary size that should be represented using arbitrary arity trees.
- Be able to use the design recipes to design with arbitrary arity trees.
- Be able to use the design recipes with mutually-referential data.
- Be able to predict and identify the correspondence between external-, self- and mutual-reference in a data definition and calls, recursion and mutual-recursion in functions that operate on the data.

Notes

- Mutually recursive data: Arbitrary-arity trees
 - Requires two cycles in the type reference graph
 - Due to arbitrary size in 2 dimensions
- Getting stuck: strategies to get unstuck
 - Make sure you have examples for what you are trying to write.
 - If you missed some examples at the beginning, GO WRITE THEM IN when you get to a situation that isn't covered!
 - Do it on paper.

Mutually-Recursive Data

- Mutually-recursive data
 - Requires two cycles in the type reference graph
 - * Due to arbitrary size in 2 dimensions
 - Data definition: do both definitions at the same time
 - Group type comments + interpretations, then put all examples and templates after
 - * <Comments + interp. for data definition 1>
 - * <Comments + interp. for data definition 2>
 - * <Examples and templates for both>
- ListOfElement
 - self-reference (SR) cycle: allows directory's list of sub-elements to be arbitrarily long
 - reference to Element: mutual reference (MR)
- Element

- reference to `ListOfElement`: mutual reference (MR)
- The **mutual reference cycle** allows each element (or node) to have an arbitrary number of sub-elements (or children)
 - i.e. allows tree to have arbitrary breadth
 - ONLY *Mutual Reference* (MR) if both types reference each other. Otherwise, it is just a reference.
- There are a few “base cases” for this tree for which it stops growing. One or more of these must be the case.
 1. When an element has non-zero data. That node cannot have children.
 2. When an element has zero data and an empty list.
 3. When an element has zero data and a list with elements with non-zero data. The element’s children will not have children (no grandchildren for you!).

HTDF for Mutually Recursive Data

- We don’t design a single function. We design a function for EACH type.
- Function naming convention: `<base-fn-name>--<data-type>`
 - All functions have a base name (eg. `sum-data`) with the type that is being operated on as a suffix (eg. `element` or `loe`)
 - eg. `sum-data--element` and `sum-data--loe`
 - All functions are named `base-fn-name` because they are **mutually recursive** & require each other to work
- Functions usually all produce the same data (but there are exceptions)
- `spd/tags` tags
 - HtDF tag at top includes all functions
 - * (`@HtDF <fn>--<type1> <fn>--<type2> ... <fn>--<typen>`)
 - Separate signatures for each function (both above purpose)
 - Separate template tags for each function (above each function)

Why does it work? Because our method is data-driven, we do all the hard work with our data definitions.

1. Well-formed, self and mutually referential type comments
2. Templates support natural mutual recursion (NMR)
3. Derived functions will
 - have the right structure, and
 - terminate in a base case

Backtracking

Three main things about backtracking:

1. Signature produces a `Type` or `false`
2. Function body of `fn. consuming ListOfX` has:
 - `(if (not (false? (find--region 1 (first lor))))))`
 - This “if not false?” pattern is important for generic functions.
 - We could instead use `region?`, but this would only work for a tree of regions.
3. Backtracking tag: `(if (not (false? is a structural characteristic of all backtracking problems`
 - Add `backtracking` to each function’s template tag
 - `(@template Region add-param backtracking)`
 - `(@template ListOfRegion add-param backtracking)`

Terminology

- Arbitrary-arity tree: nodes can have an arbitrary number of children
 - Arbitrarily deep: an unknown number of levels
 - Arbitrarily “wide”: an unknown number of children
- Mutual Reference: structure in types
- Mutual Recursion: structure in templates
- Natural Mutual Recursion: structure in function

Reference, self-reference, and mutual-reference terms

Cause and effect of template rules (from top to bottom)

- Referential Data:
 1. Reference (R) in type comment
 2. Natural Helper (NH) in template
 3. Helper function wraps type causing NH in function
- Self-Referential Data:
 1. Self-reference (SR) in type comment
 2. Natural recursion (NR) in template
 3. Helper function wraps type causing NR in function
- Mutually-Recursive Data:
 1. Mutual Reference Cycle (MR) in type comments
 2. Natural Mutual Recursion (MR) in templates
 3. Helper function wraps type causing MR in function

Off-topic Questions

In Racket, For a search function, we can produce `Value` or `false` which represents two cases:

1. Success! We find the key and produce its value.
2. Failure. We do not find the key and return `false`.

How might we transfer this to other languages like C++ or Java?

In C++, there is a `find` function for vector lists. It behaves as such:

1. If found element, returns iterator to it.
2. Otherwise, return iterator to the last element.

Is there a better way? I have often found myself being unsure of how to represent a “failing” case. The solutions I have used are either,

1. Return 0, -1, or some other meaningless value. Give this value meaning where the function is called (i.e. user is responsible for implementation details)
2. Use an Enum to give meaning to arbitrary integers.
3. Throw an exception. (I assume this is the best method)