CSci 2041
Advanced Programming Principles
L3: Recursive computations over recursive data

Prof. Eric Van Wyk

Fall 2014

Complex data is often hierarchical and recursive.

- Binary trees have leaf nodes and nodes that contain other trees
- Program expressions may be constants or variables but may also be operations that contain other expressions

Processing this sort of data begs for recursion.

While-loops are linear in nature and don't fit the needs of this richer kind of data.

Types play a key role in describing, defining, and processing this data, so we'll review some key notions from types first.

Recall ...

- OCaml has a strong static type system
- C has a weak static type system
- Python has a strong (?) dynamic type system

An important question about languages is to what degree can programmers create their own types and their corresponding values.

Base types and values

- Languages almost always have some collection of base or primitive types.
  - int
  - float
  - char
- They may correspond to hardware supported values and operations.
- A corresponding set of base or primitive values can be specified in the language. For example, one can often write the following:
  - `1` or `42`
  - `1.219`
  - `'x'`

More interesting types

Here are a few more interesting types and values.

- In OCaml:
  - `int list` — `[1; 2; 3]`
  - `string list` — `["Hello"; "World"]`
  - `int int list` — `[ [1;2;3]; [4;5;6] ]`
  - `int * string` — `(5, "five")`
- In Java:
  - *e.g.* `Person`, a class
  - `ArrayList<Integer>`, an instantiated parameterized class
- In C:
  - `int []`
  - `float *`
  - `enum {Monday, Wednesday, Friday}`
  - `struct { float x; float y;}`

Programmer defined types

- Languages almost always provide a way to create new types (and their corresponding values).
- These new types are often constructed from existing types.
- These new types have corresponding new values.
- Some terms we'll be using:
  - type
  - type constructor
  - value
  - value constructor

  Let's consider how do these apply to some OCaml lists.

OCaml lists: types

- Consider the type `int list`
- `int` is a type
- What is `list`?
- `list` is a type constructor
- We use `list` to construct new types.
- Think of `list` as a (postfix) function that takes a type and returns a type.
- For example
  - `list` applied to `int` gives `int list`
  - `list` applied to `string` gives `string list`
  - `list` applied to `int list` gives `int list list`

OCaml lists: values

- What about values?
- `1 :: 2 :: 3 :: []` is a value
  - it is a list of three integers
- `::` is a value constructor
- `[]` is a value constructor

What follows is a series of example data types from various languages that we will represent using abstract data types.

C enumerated types

- Consider `enum color {red, green, yellow };`
- `color` is a new type
- We have enumerated all the values of this type.
- The values are `red`, `green`, and `yellow`.

OOP classes

- Consider a `Shape` abstract class with concrete subclasses `Circle`, `Square`, `Triangle`
- These are all new types.
- Values are object of these types
- We can think of constructor methods as value constructors.

OOP classes - parameterized

- Consider the type `ArrayList<Integer>`
- `ArrayList` is a programmer defined type constructor. Specifically, a parameterized class.
- `ArrayList` is applied to `Integer` to create the new type `ArrayList<Integer>`.
- As before, objects of type `ArrayList<Integer>`, for example, are values
- Constructor methods are value constructors.

Recap

- Type
- Type constructor
- Value
- Value constructor

In discussing programming techniques, we may ask what kinds of types, type constructors, values, and value constructors can be defined in the language we are using?

What about `class` and `enum`, are they type constructors? We'll say no. They are parts of a language. They cannot be defined in the language.

Disjoint Unions

Disjoint unions are an expressive means for creating new types, type constructors, values, and value constructors.

Many of the types described above can be expressed using disjoint unions.

OCaml disjoint union examples

- colors - as in C enumerated types
- shapes - for variants on a type
- a `maybe` type for optional values
- lists
- trees

Note: types and type constructors must begin with a lowercase letter, while value constructors must begin with an uppercase letter.

The samples developed in class will be in the utop histories and in the `code-examples` directory of the public repository.

- Enumerated types, as found in C, can be easily represented using disjoint unions.
- For example,
  `type color = Red | Green | Yellow`
- This defines the type `color` with values (or nullary value constructors) `Red`, `Green`, and `Yellow`.

Pattern matching

- Can we write a function called `color_to_string` that has type `color -> string` ?
- Let's do that in OCaml and test it out.

Shapes

Recall the shapes OOP example with abstract class `Shape` and concrete subclasses `Circle`, `Square`, `Triangle`.

- As an disjoint union we'll have a type `shape` and value constructors `Circle`, `Square`, `Triangle`.
- Let's define this type
- Value constructors can be followed by "`of` *type*"
- They take values (of the specified type) to create new values.
- So `Red` is a nullary value constructor.
- Value constructors are not curried, they take 0 or 1 argument. But that argument may be a tuple of several values.

Pattern matching on this type

- Lets sketch a function `sum_perimeters` that takes a list of shapes and returns the sum of their perimeters.

"maybe"

- A common idiom in languages with pointers is for a function to return a null pointer if the desired result could not be computed, and a pointer to the result if it could be computed.
- But this is unsafe - it is too easy to forget to check the pointer and to access it.
- We can build a "optional" type using disjoint unions.
- `type 'a = Nothing | Just of 'a`
- Let's rewrite `sum_diffs` to use this type.

OCaml provided disjoint unions

- `option` - replaces out `maybe` type, which is just taken from Haskell.
  ```
  type 'a option = Some of 'a | None
  ```
- Lists
  `[]` and `::` are value constructors for the type `'a list`.
- We can define our own lists
  ```
  type 'a myList = Nil | Cons of ('a * 'a
  myList)
  ```

Recursive datatypes

Lists are one example of "recursive data"

- List values are constructed from list values
- Specifically,
  1. lists are a special empty list value or
  2. they are constructed from an element and another list value

Values of a recursive type are constructed from other values of that same (recursive) type.
Like recursive functions, there are

- "base cases" in the type definition, and
- "recursive cases"

Examples of recursive datatypes

- Lists - both our version and the OCaml-provided version
- Binary trees.
  - trees are leaves, or
  - nodes containing other trees

  (In addition, values may also decorate these nodes.)
- Expressions in programs.
  - leaves are constants or variables
  - addition and multiplication expressions are trees with child expression trees

Disjoint union definitions

```
type [ ⟨type variables⟩ ]  ⟨type-name⟩ =
  |  ⟨ValueConstructor⟩ [ of ⟨type⟩ [ * ⟨type⟩ ] ]
  |  ⟨ValueConstructor⟩ [ of ⟨type⟩ [ * ⟨type⟩ ] ]
  | ...
```

The type being defined (on the left of $=$) can be used on the right, in a recursive case.

Computing over recursive data

Computations over recursive data are typically structured as follows:

- some simple computation for base cases
- perform the computation on the recursive components of the data, then combine the resulting values in an appropriate way

Recall, `sum` over an `int list`

```
let rec sum xs =
  match xs with
  | [ ] -> 0
  | x::rest -> x + sum rest
```

This computation has this structure.

Kinds of types

- Tuples, as we've seen are product types.
- Their associated values can be seen as the Cartesian product of the values of their component types.
- Records, structs, and to some extent objects, are also product types.
- For example `int * string * char list`

Kinds of types

- Sum types, at first glance, can be seen as the union of the values of their component types.
- disjoint unions are sometimes called "discriminated unions" or "tagged unions".
- They are sum types that have a tag (the value constructor) associated with each value.
- For example, `type number = IntVal of int | FloatVal of float`