OCaml Datatypes Part II: An Exercise in Type Design

COS 326

David Walker

Princeton University

Example Type Design

IBM developed GML (Generalize Markup Language) in 1969

- http://en.wikipedia.org/wiki/IBM_Generalized_Markup_Language
- Precursor to SGML, HTML and XML

```
:h1.Chapter 1: Introduction
:p.GML supported hierarchical containers, such as
:ol
:li.Ordered lists (like this one),
:li.Unordered lists, and
:li.Definition lists
:eol.
as well as simple structures.
:p.Markup Minimization (later generalized and
formalized in SGML), allowed the end-tags to be
omitted for the “h1” and “p” elements.
```

Simplified GML

To process a GML document, an OCaml program would:

- Read a series of characters from a text file & Parse GML structure
- Represent the information content as an OCaml data structure
- Analyze or transform the data structure
- Print/Store/Communicate results

We will focus on how to *represent* and *transform* the information content of a GML document.

Example Type Design

- A GML document consists of:
  - a list of elements
- An element is either:
  - a word or markup applied to an element
- Markup is either:
  - italicize, bold, or a font name

Example Type Design

- A GML document consists of:
  - a list of elements
- An element is either:
  - a word or markup applied to an element
- Markup is either:
  - italicize, bold, or a font name

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

Example Data


```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

```
let d = [ Formatted (Bold,
            Formatted (Font "Arial",
              Words ["Chapter";"One"]));

          Words ["It"; "was"; "a"; "dark";
                 "&"; "stormy; "night."; "A"];

          Formatted (Ital, Words["shot"]);

          Words ["rang"; "out."] ];;
```

Challenge

- Change all of the "Arial" fonts in a document to "Courier".
- Of course, when we program functionally, we implement *change* via a function that
  - receives one data structure as input
  - builds a new (different) data structure as an output

Challenge

- Change all of the “Arial” fonts in a document to “Courier”.

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

Challenge

- Change all of the "Arial" fonts in a document to "Courier".

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

- Technique: approach the problem top down, work on doc first:

```
let rec chfonts (elts:doc) : doc =
```

Challenge

- Change all of the "Arial" fonts in a document to "Courier".

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

- Technique: approach the problem top down, work on doc first:

```
let rec chfonts (elts:doc) : doc =
  match elts with
  | [] ->
  | hd::tl ->
```

Challenge

- Change all of the "Arial" fonts in a document to "Courier".

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

- Technique: approach the problem top down, work on doc first:

```
let rec chfonts (elts:doc) : doc =
  match elts with
  | [] -> []
  | hd::tl -> (chfont hd)::(chfonts tl)
```

Changing fonts in an element

- Change all of the "Arial" fonts in a document to "Courier".

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

- Next work on changing the font of an element:

```
let rec chfont (e:elt) : elt =
```

Changing fonts in an element

- Change all of the "Arial" fonts in a document to "Courier".

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

- Next work on changing the font of an element:

```
let rec chfont (e:elt) : elt =
  match e with
   | Words ws ->
   | Formatted(m,e) ->
```

Changing fonts in an element

- Change all of the "Arial" fonts in a document to "Courier".

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

- Next work on changing the font of an element:

```
let rec chfont (e:elt) : elt =
  match e with
   | Words ws -> Words ws
   | Formatted(m,e) ->
```

Changing fonts in an element

- Change all of the "Arial" fonts in a document to "Courier".

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

- Next work on changing the font of an element:

```
let rec chfont (e:elt) : elt =
  match e with
   | Words ws -> Words ws
   | Formatted(m,e) -> Formatted(chmarkup m, chfont e)
```

Changing fonts in an element

- Change all of the "Arial" fonts in a document to "Courier".

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

- Next work on changing a markup:

```
let chmarkup (m:markup) : markup =
```

Changing fonts in an element

- Change all of the "Arial" fonts in a document to "Courier".

```
type markup = Ital | Bold | Font of string

type elt =
  Words of string list
| Formatted of markup * elt

type doc = elt list
```

- Next work on changing a markup:

```
let chmarkup (m:markup) : markup =
  match m with
  | Font "Arial" -> Font "Courier"
  | _ -> m
```

Summary: Changing fonts in an element

- Change all of the "Arial" fonts in a document to "Courier"
- Lesson: function structure follows type structure

```
let chmarkup (m:markup) : markup =
  match m with
   | Font "Arial" -> Font "Courier"
   | _ -> m

let rec chfont (e:elt) : elt =
  match e with
   | Words ws -> Words ws
   | Formatted(m,e) -> Formatted(chmarkup m, chfont e)

let rec chfonts (elts:doc) : doc =
  match elts with
   | [] -> []
   | hd::tl -> (chfont hd)::(chfonts tl)
```

Poor Style

- Consider again our definition of markup and markup change:

```
type markup =
  Ital | Bold | Font of string


let chmarkup (m:markup) : markup =
  match m with
  | Font "Arial" -> Font "Courier"
  | _ -> m
```

Poor Style

- What if we make a change:

```
type markup =
  Ital | Bold | Font of string | TTFont of string


let chmarkup (m:markup) : markup =
  match m with
   | Font "Arial" -> Font "Courier"
   | _ -> m
```

the underscore silently catches all possible alternatives

this may not be what we want -- perhaps there is an Arial TT font

it is better if we are alerted of all functions whose implementation may need to change

Better Style

- Original code:

```
type markup =
  Ital | Bold | Font of string

let chmarkup (m:markup) : markup =
  match m with
  | Font "Arial" -> Font "Courier"
  | Ital | Bold -> m
```

Better Style

- Updated code:

```
type markup =
  Ital | Bold | Font of string | TTFont of string

let chmarkup (m:markup) : markup =
  match m with
   | Font “Arial” -> Font “Courier”
   | Ital | Bold -> m
```

```
..match m with
    | Font "Arial" -> Font "Courier"
    | Ital | Bold -> m..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
TTFont _
```

Better Style

- Updated code, fixed:

```
type markup =
  Ital | Bold | Font of string | TTFont of string

let chmarkup (m:markup) : markup =
  match m with
   | Font "Arial" -> Font "Courier"
   | TTFont "Arial" -> TTFont "Courier"
   | Font s -> Font s
   | TTFont s -> TTFont s
   | Ital | Bold -> m
```

- Lesson: use the type checker where possible to help you maintain your code

A couple of practice problems

- Write a function that gets rid of immediately redundant markup in a document.
  - Formatted(Ital, Formatted(Ital,e)) can be simplified to Formatted(Ital,e)
  - write maps and folds over markups
- Design a datatype to describe bibliography entries for publications. Some publications are journal articles, others are books, and others are conference papers. Journals have a name, number and issue; books have an ISBN number; All of these entries should have a title and author.
  - design a sorting function
  - design maps and folds over your bibliography entries

To Summarize

- Design recipe for writing OCaml code:
  - write down English specifications
    - try to break problem into obvious sub-problems
  - write down some sample test cases
  - write down the signature (types) for the code
  - use the signature to guide construction of the code:
    - tear apart inputs using pattern matching
      - make sure to cover all of the cases! (OCaml will tell you)
    - handle each case, building results using data constructor
      - this is where human intelligence comes into play
      - the "skeleton" given by types can almost be done automatically!
    - clean up your code
  - use your sample tests (and ideally others) to ensure correctness