Functional Programming

by Łukasz Stafiniak

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

Lecture 2: Algebra

Algebraic Data Types and some curious analogies

A Glimpse at Type Inference

For a refresher, let's try to use the rules we introduced last time on some simple examples. Starting with `fun x -> x`. $[?]$ will mean "dunno yet".

$$\frac{[?]}{\texttt{fun x -> x}\colon [?]}$$

use $\to$ introduction:

$$\frac{\dfrac{}{\texttt{x}\colon a}\,x}{\texttt{fun x -> x}\colon [?] \to [?]}$$

$\dfrac{}{\texttt{x}\colon a}\,x$ matches with $\dfrac{}{x\colon a}\,x$ since $e = \texttt{x}$

$$\begin{array}{c} x\colon a \\ | \\ e\colon b \end{array}$$

$$\frac{\dfrac{}{\texttt{x}\colon a}\,x}{\texttt{fun x -> x}\colon a \to a}$$

since $b = a$ because $x\colon a$ matched with $e\colon b$

Because a is arbitrary, OCaml puts a *type variable* `'a` for it:

```
# fun x -> x;;
- : 'a -> 'a = <fun>
```

Let's try `fun x -> x+1`, which is the same as `fun x -> ((+) x) 1` (try it with OCaml/F#!). $[?\alpha]$ will mean "dunno yet, but the same as in other places with $[?\alpha]$".

$$\dfrac{[?]}{\texttt{fun x -> ((+) x) 1}:[?]}$$

use $\to$ introduction:

$$\dfrac{\dfrac{[?]}{\texttt{((+) x) 1}:[?\alpha]}}{\texttt{fun x -> ((+) x) 1}:[?]\to[?\alpha]}$$

use $\to$ elimination:

$$\dfrac{\dfrac{\dfrac{[?]}{\texttt{(+) x}:[?\beta]\to[?\alpha]}\quad\dfrac{[?]}{\texttt{1}:[?\beta]}}{\texttt{((+) x) 1}:[?\alpha]}}{\texttt{fun x -> ((+) x) 1}:[?]\to[?\alpha]}$$

we know that $\texttt{1}:\texttt{int}$

$$\dfrac{\dfrac{\dfrac{[?]}{\texttt{(+) x}:\texttt{int}\to[?\alpha]}\quad\dfrac{}{\texttt{1}:\texttt{int}}\,(\text{constant})}{\texttt{((+) x) 1}:[?\alpha]}}{\texttt{fun x -> ((+) x) 1}:[?]\to[?\alpha]}$$

application again:

$$\dfrac{\dfrac{\dfrac{\dfrac{[?]}{\texttt{(+)}:[?\gamma]\to\texttt{int}\to[?\alpha]}\quad\dfrac{[?]}{\texttt{x}:[?\gamma]}}{\texttt{(+) x}:\texttt{int}\to[?\alpha]}\quad\dfrac{}{\texttt{1}:\texttt{int}}\,(\text{constant})}{\texttt{((+) x) 1}:[?\alpha]}}{\texttt{fun x -> ((+) x) 1}:[?]\to[?\alpha]}$$

it's our `x`!

$$\dfrac{\dfrac{\dfrac{}{\texttt{(+)}:[?\gamma]\to\texttt{int}\to[?\alpha]}[?] \quad \dfrac{}{\texttt{x}:[?\gamma]}\texttt{x}}{\texttt{(+) x}:\texttt{int}\to[?\alpha]} \quad \dfrac{}{\texttt{1}:\texttt{int}}(\text{constant})}{\texttt{((+) x) 1}:[?\alpha]}}{\texttt{fun x -> ((+) x) 1}:[?\gamma]\to[?\alpha]}$$

but $\texttt{(+)}:\texttt{int}\to\texttt{int}\to\texttt{int}$

$$\dfrac{\dfrac{\dfrac{}{\texttt{(+)}:\texttt{int}\to\texttt{int}\to\texttt{int}}(\text{constant}) \quad \dfrac{}{\texttt{x}:\texttt{int}}\texttt{x}}{\texttt{(+) x}:\texttt{int}\to\texttt{int}} \quad \dfrac{}{\texttt{1}:\texttt{int}}(\text{constant})}{\texttt{((+) x) 1}:\texttt{int}}}{\texttt{fun x -> ((+) x) 1}:\texttt{int}\to\texttt{int}}$$

Curried form

When there are several arrows "on the same depth" in a function type, it means that the function returns a function: e.g. `(+): int→int→int` is just a shorthand for `(+): int→(int→int)`. It is very different from

$$\texttt{fun f -> (f 1) + 1:(int→int)→int}$$

For addition, instead of `(fun x -> x+1)` we can write `((+) 1)`. What expanded form does `((+) 1)` correspond to exactly (computationally)?

We will get used to functions returning functions when learning about the *lambda calculus*.

Algebraic Data Types

- Last time we learned about the `unit` type, variant types like:

  ```
  type int_string_choice = A of int | B of string
  ```

  and also tuple types, record types, and type definitions.

- Variants don't have to have arguments: instead of `A of unit` just use `A`.
  - In OCaml, variants take multiple arguments rather than taking tuples as arguments: `A of int * string` is different than `A of (int * string)`. But it's not important unless you get bitten by it.

- Type definitions can be recursive!

  ```
  type int_list = Empty | Cons of int * int_list
  ```

  Let's see what we have in `int_list`:
  `Empty`, `Cons (5, Cons (7, Cons (13, Empty)))`, etc.

- Type `bool` can be seen as `type bool = true | false`, type `int` can be seen as a very large `type int = 0 | -1 | 1 | -2 | 2 | ...`

- Type definitions can be *parametric* with respect to types of their components (more on this in lecture about polymorphism), for example a list elements of arbitrary type:

  ```
  type 'elem list = Empty | Cons of 'elem * 'elem list
  ```

  - Type variables must start with `'`, but since OCaml will not remember the names we give, it's customary to use the names OCaml uses: `'a`, `'b`, `'c`, `'d`...
  - The syntax in OCaml is a bit strange: in F# we write `list<'elem>`. OCaml syntax mimics English, silly example:

    ```
    type 'white_color dog = Dog of 'white_color
    ```

  - With multiple parameters:
    - OCaml:
      ```
      type ('a, 'b) choice = Left of 'a | Right of 'b
      ```
    - F#:
      ```
      type choice<'a,'b> = Left of 'a | Right of 'b
      ```
    - Haskell:
      ```
      data Choice a b = Left a | Right b
      ```

Syntactic Bread and Sugar

- Names of variants, called *constructors*, must start with capital letter – so if we wanted to define our own booleans, it would be

  ```
  type my_bool = True | False
  ```

  Only constructors and module names can start with capital letter.

  - *Modules* are "shelves" with values. For example, `List` has operations on lists, like `List.map` and `List.filter`.
- Did I mention that we can use `record.field` to access a field?
- `fun x y -> e` stands for `fun x -> fun y -> e`, etc. – and of course, `fun x -> fun y -> e` parses as `fun x -> (fun y -> e)`
- `function A x -> e1 | B y -> e2` stands for `fun p -> match p with A x -> e1 | B y -> e2`, etc.
  - the general form is: `function` *PATTERN-MATCHING* stands for `fun` v `-> match` v `with` *PATTERN-MATCHING*
- `let f` ARGS `= e` is a shorthand for `let f = fun` ARGS `-> e`

Pattern Matching

- Recall that we introduced `fst` and `snd` as means to access elements of a pair. But what about bigger tuples? The "basic" way of accessing any tuple reuses the `match` construct. Functions `fst` and `snd` can easily be defined!

```
let fst = fun p -> match p with (a, b) -> a
let snd = fun p -> match p with (a, b) -> b
```

- It also works with records:

```
type person = {name: string; surname: string; age: int}
match {name="Walker"; surname="Johnnie"; age=207}
with {name=n; surname=sn; age=a} -> "Hi "^sn^"!"
```

- The left-hand-sides of `->` in `match` expressions are called **patterns**.
- Patterns can be nested:

```
match Some (5, 7) with None -> "sum: nothing"
  | Some (x, y) -> "sum: " ^ string_of_int (x+y)
```

- A pattern can just match the whole value, without performing destructuring: `match f x with v ->...` is the same as `let v = f x in ...`
- When we do not need a value in a pattern, it is good practice to use the underscore: `_` (which is not a variable!)

```
let fst (a,_) = a
let snd (_,b) = b
```

- A variable can only appear once in a pattern (it is called *linearity*).
- But we can add conditions to the patterns after `when`, so linearity is not really a problem!

```
match p with (x, y) when x = y -> "diag" | _ -> "off-diag"
```

```
let compare a b = match a, b with
  | (x, y) when x < y -> -1
  | (x, y) when x = y -> 0
  | _ -> 1
```

- We can skip over unused fields of a record in a pattern.
- We can compress our patterns by using | inside a single pattern:

```
type month =
  | Jan | Feb | Mar | Apr | May | Jun
  | Jul | Aug | Sep | Oct | Nov | Dec
type weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
type date =
  {year: int; month: month; day: int; weekday: weekday}
let day =
  {year = 2012; month = Feb; day = 14; weekday = Wed};;
match day with
  | {weekday = Sat | Sun} -> "Weekend!"
  | _ -> "Work day"
```

- We use `(pattern as v)` to name a nested pattern:

```
match day with
  | {weekday = (Mon | Tue | Wed | Thu | Fri as wday)}
      when not (day.month = Dec && day.day = 24) ->
    Some (work (get_plan wday))
  | _ -> None
```

Interpreting Algebraic DTs as Polynomials

Let's do a peculiar translation: take a data type and replace | with $+$, * with $\times$, treating record types as tuple types (i.e. erasing field names and translationg ; as $\times$).

There is a special type for which we cannot build a value:

```
type void
```

(yes, it is its definition, no = `something` part). Translate it as 0.

Translate the `unit` type as 1. Since variants without arguments behave as variants `of unit`, translate them as 1 as well. Translate `bool` as 2.

Translate `int`, `string`, `float`, type parameters and other types of interest as variables. Translate defined types by their translations (substituting variables if necessary).

Give name to the type being defined (denoting a function of the variables introduced). Now interpret the result as ordinary numeric polynomial! (Or "rational function" if it is recursively defined.)

Let's have fun with it.

```
type date = {year: int; month: int; day: int}
```

$$D = x\,x\,x = x^3$$

```
type 'a option = None | Some of 'a   (* built-in type *)
```

$$O = 1 + x$$

```
type 'a my_list = Empty | Cons of 'a * 'a my_list
```

$$L = 1 + x\,L$$

```
type btree = Tip | Node of int * btree * btree
```

$$T = 1 + x\,TT = 1 + x\,T^2$$

When translations of two types are equal according to laws of high-school algebra, the types are *isomorphic*, that is, there exist 1-to-1 functions from one type to the other.

Let's play with the type of binary trees:

$$\begin{aligned} T &= 1 + x\,T^2 = 1 + x\,T + x^2\,T^3 = 1 + x + x^2\,T^2 + x^2\,T^3 = \\ &= 1 + x + x^2\,T^2\,(1+T) = 1 + x(1 + x\,T^2\,(1+T)) \end{aligned}$$

Now let's translate the resulting type:

```
type repr =
  (int * (int * btree * btree * btree option) option) option
```

Try to find the isomorphism functions `iso1` and `iso2`

```
val iso1 : btree -> repr
val iso2 : repr -> btree
```

i.e. functions such that for all trees `t`, `iso2 (iso1 t) = t`, and for all representations `r`, `iso1 (iso2 r) = r`.

My first failed attempt:

```
# let iso1 (t : btree) : repr =
  match t with
    | Tip -> None
    | Node (x, Tip, Tip) -> Some (x, None)
    | Node (x, Node (y, t1, t2), Tip) ->
      Some (x, Some (y, t1, t2, None))
    | Node (x, Node (y, t1, t2), t3) ->
      Some (x, Some (y, t1, t2, Some t3));;
            Characters 32-261: [...]
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Node (_, Tip, Node (_, _, _))
```

I forgot about one case. It seems difficult to guess the solution, have you found it on your try?

Let's divide the task into smaller steps corresponding to selected intermediate points in the transformation of the polynomial:

```
type ('a, 'b) choice = Left of 'a | Right of 'b
type interm1 =
  ((int * btree, int * int * btree * btree * btree) choice)
  option
type interm2 =
  ((int, int * int * btree * btree * btree option) choice)
  option

let step1r (t : btree) : interm1 =
  match t with
    | Tip -> None
    | Node (x, t1, Tip) -> Some (Left (x, t1))
    | Node (x, t1, Node (y, t2, t3)) ->
      Some (Right (x, y, t1, t2, t3))
```

```
let step2r (r : interm1) : interm2 =
  match r with
    | None -> None
    | Some (Left (x, Tip)) -> Some (Left x)
    | Some (Left (x, Node (y, t1, t2))) ->
      Some (Right (x, y, t1, t2, None))
    | Some (Right (x, y, t1, t2, t3)) ->
      Some (Right (x, y, t1, t2, Some t3))

let step3r (r : interm2) : repr =
  match r with
    | None -> None
    | Some (Left x) -> Some (x, None)
    | Some (Right (x, y, t1, t2, t3opt)) ->
      Some (x, Some (y, t1, t2, t3opt))

let iso1 (t : btree) : repr =
  step3r (step2r (step1r t))
```

Define `step1l`, `step2l`, `step3l`, and `iso2`. Hint: now it's trivial!

Take-home lessons:

- Try to define data structures so that only information that makes sense can be represented – as long as it does not overcomplicate the data structures. Avoid catch-all clauses when defining functions. The compiler will then tell you if you have forgotten about a case.
- Divide solutions into small steps so that each step can be easily understood and checked.

Differentiating Algebraic Data Types

Of course, you would say, the pompous title is wrong, we will differentiate the translated polynomials. But what sense does it make?

It turns out, that taking the partial derivative of a polynomial resulting from translating a data type, gives us, when translated back, a type representing how to change one occurrence of a value of type corresponding to the variable with respect to which we computed the partial derivative.

Take the "date" example:

```
type date = {year: int; month: int; day: int}
```

$$\begin{aligned} D &= x\,x\,x = x^3 \\ \frac{\partial D}{\partial x} &= 3\,x^2 = x\,x + x\,x + x\,x \end{aligned}$$

(we could have left it at $3\,x\,x$ as well). Now we construct the type:

```
type date_deriv =
  Year of int * int | Month of int * int | Day of int * int
```

Now we need to introduce and use ("eliminate") the type `date_deriv`.

```
let date_deriv {year=y; month=m; day=d} =
  [Year (m, d); Month (y, d); Day (y, m)]

let date_integr n = function
  | Year (m, d) -> {year=n; month=m; day=d}
  | Month (y, d) -> {year=y; month=n; day=d}
  | Day (y, m) -> {year=y; month=m, day=n}
;;
List.map (date_integr 7)
  (date_deriv {year=2012; month=2; day=14})
```

Let's do now the more difficult case of binary trees:

```
type btree = Tip | Node of int * btree * btree
```

$$\begin{aligned} T &= 1 + x\,T^2 \\ \frac{\partial T}{\partial x} &= 0 + T^2 + 2\,x\,T\,\frac{\partial T}{\partial x} = TT + 2\,x\,T\,\frac{\partial T}{\partial x} \end{aligned}$$

(again, we could expand further into $\frac{\partial T}{\partial x} = TT + x\,T\,\frac{\partial T}{\partial x} + x\,T\,\frac{\partial T}{\partial x}$).

Instead of translating 2 as `bool`, we will introduce new type for clarity:

```
type btree_dir = LeftBranch | RightBranch
type btree_deriv =
  | Here of btree * btree
  | Below of btree_dir * int * btree * btree_deriv
```

(You might someday hear about *zippers* – they are "inverted" w.r.t. our type, in zippers the hole comes first.)

Write a function that takes a number and a `btree_deriv`, and builds a `btree` by putting the number into the "hole" in `btree_deriv`.

Solution:

```
let rec btree_integr n =
  | Here (ltree, rtree) -> Node (n, ltree, rtree)
  | Below (LeftBranch, m, rtree) ->
    Node (m, btree_integr n ltree, rtree)
  | Below (RightBranch, m, ltree) ->
    Node (m, ltree, btree_integr n rtree)
```

Homework

Write a function `btree_deriv_at` that takes a predicate over integers (i.e. a function `f: int -> bool`), and a `btree`, and builds a `btree_deriv` whose "hole" is in the first position for which the predicate returns true. It should actually return a `btree_deriv option`, with `None` in case the predicate does not hold for any node.

*This homework is due for the class **after** the Computation class, i.e. for (before) the Functions class.*