Functional Programming

by Łukasz Stafiniak

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

Lecture 5: Polymorphism & ADTs

Parametric types. Abstract Data Types.

Example: maps using red-black trees.

If you see any error on the slides, let me know!

Type Inference

We have seen the rules that govern the assignment of types to expressions, but how does OCaml guess what types to use, and when no correct types exist? It solves equations.

- Variables play two roles: of *unknowns* and of *parameters*.
  - Inside:

    ```
    # let f = List.hd;;
    val f : 'a list -> 'a
    ```

    'a is a parameter: it can become any type. Mathematically we write: $f : \forall\alpha.\alpha\,\mathrm{list} \to \alpha$ – the quantified type is called a *type scheme*.
  - Inside:

    ```
    # let x = ref [];;
    val x : '_a list ref
    ```

    '_a is an unknown. It stands for a particular type like `float` or `(int -> int)`, OCaml just doesn't yet know the type.

- OCaml only reports unknowns like `'_a` in inferred types for reasons not relevant to functional programming. When unknowns appear in inferred type against our expectations, *η-expansion* may help: writing `let f x = expr x` instead of `let f = expr` – for example:

```
# let f = List.append [];;
val f : '_a list -> '_a list = <fun>
# let f l = List.append [] l;;
val f : 'a list -> 'a list = <fun>
```

- A *type environment* specifies what names (corresponding to parameters and definitions) are available for an expression, because they were introduced above it, and it specifies their types.
- Type inference solves equations over unknowns. "What has to hold so that $e : \tau$ in type environment Γ?"
  - If, for example, $f : \forall\alpha.\alpha\ \text{list} \to \alpha \in \Gamma$, then for $f : \tau$ we introduce $\gamma\ \text{list} \to \gamma = \tau$ for some fresh unknown γ.
  - For $e_1\ e_2 : \tau$ we introduce $\beta = \tau$ and ask for $e_1 : \gamma \to \beta$ and $e_2 : \gamma$, for some fresh unknowns β, γ.

- For $\mathrm{fun}\, x \to e\colon \tau$ we introduce $\beta \to \gamma = \tau$ and ask for $e\colon \gamma$ in environment $\{x\colon \beta\} \cup \Gamma$, for some fresh unknowns β, γ.
- Case $\mathrm{let}\, x = e_1\, \mathrm{in}\, e_2\colon \tau$ is different. One approach is to *first* solve the equations that we get by asking for $e_1\colon \beta$, for some fresh unknown β. Let's say a solution $\beta = \tau_\beta$ has been found, $\alpha_1 ... \alpha_n \beta_1 ... \beta_m$ are the remaining unknowns in τ_β, and $\alpha_1 ... \alpha_n$ are all that do not appear in Γ. Then we ask for $e_2\colon \tau$ in environment $\{x\colon \forall \alpha_1 ... \alpha_n . \tau_\beta\} \cup \Gamma$.
- Remember that whenever we establish a solution $\beta = \tau_\beta$ to an unknown β, it takes effect everywhere!
- To find a type for e (in environment Γ), we pick a fresh unknown β and ask for $e\colon \beta$ (in Γ).

- The "top-level" definitions for which the system infers types with variables are called *polymorphic*, which informally means "working with different shapes of data".
  - This kind of polymorphism is called *parametric polymorphism*, since the types have parameters. A different kind of polymorphism is provided by object-oriented programming languages.

Parametric Types

- Polymorphic functions shine when used with polymorphic data types. In:

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

  we define lists that can store elements of any type `'a`. Now:

  ```
  # let tail l =
    match l with
      | Empty -> invalid_arg "tail"
      | Cons (_, tl) -> tl;;
        val tail : 'a my_list -> 'a my_list
  ```

  is a polymorphic function: works for lists with elements of any type.

- A *parametric type* like `'a my_list` *is not* itself a data type but a family of data types: `bool my_list`, `int my_list` etc. *are* different types.

  - We say that the type `int my_list` *instantiates* the parametric type `'a my_list`.

- In OCaml, the syntax is a bit confusing: type parameters precede type name. For example:

  ```
  type ('a, 'b) choice = Left of 'a | Right of 'b
  ```

  has two parameters. Mathematically we would write $\mathrm{choice}(\alpha, \beta)$.

  - Functions do not have to be polymorphic:

    ```
    # let get_int c =
      match c with
        | Left i -> i
        | Right b -> if b then 1 else 0;;
          val get_int : (int, bool) choice -> int
    ```

- In F#, we provide parameters (when more than one) after type name:

  ```
  type choice<'a,'b> = Left of 'a | Right of 'b
  ```

- In Haskell, we provide type parameters similarly to function arguments:

  ```
  data Choice a b = Left a | Right b
  ```

Type Inference, Formally

- A statement that an expression has a type in an environment is called a *type judgement*. For environment $\Gamma = \{x\colon \forall\alpha_1 ... \alpha_n.\tau_x; \ ...\}$, expression e and type τ we write

$$\Gamma \vdash e\colon \tau$$

- We will derive the equations in one go using $[\![\cdot]\!]$, to be solved later. Besides equations we will need to manage introduced variables, using existential quantification.
- For local definitions we require to remember what constraints should hold when the definition is used. Therefore we extend *type schemes* in the environment to: $\Gamma = \{x\colon \forall\beta_1 ... \beta_m[\exists\alpha_1 ... \alpha_n.D].\tau_x; \ ...\}$ where D are equations – keeping the variables $\alpha_1 ... \alpha_n$ introduced while deriving D in front.
  - A simpler form would be enough: $\Gamma = \{x\colon \forall\beta[\exists\alpha_1 ... \alpha_n.D].\beta; \ ...\}$

$$
\begin{array}{rcl}
[\![\Gamma \vdash x\colon \tau]\!] & = & \exists \bar{\beta}'\bar{\alpha}'.(D[\bar{\beta}\bar{\alpha} := \bar{\beta}'\bar{\alpha}'] \wedge \tau_x[\bar{\beta}\bar{\alpha} := \bar{\beta}'\bar{\alpha}'] \doteq \tau) \\
 & & \text{where } \Gamma(x) = \forall \bar{\beta}[\exists \bar{\alpha}.D].\tau_x,\ \bar{\beta}'\bar{\alpha}' \# \mathrm{FV}(\Gamma, \tau) \\
 & & \\
[\![\Gamma \vdash \mathbf{fun}\ x \text{->} e\colon \tau]\!] & = & \exists \alpha_1\alpha_2.([\![\Gamma\{x\colon \alpha_1\} \vdash e\colon \alpha_2]\!] \wedge \alpha_1 \to \alpha_2 \doteq \tau), \\
 & & \text{where } \alpha_1\alpha_2 \# \mathrm{FV}(\Gamma, \tau) \\
 & & \\
[\![\Gamma \vdash e_1\, e_2\colon \tau]\!] & = & \exists \alpha.([\![\Gamma \vdash e_1\colon \alpha \to \tau]\!] \wedge [\![\Gamma \vdash e_2\colon \alpha]\!]), \alpha \# \mathrm{FV}(\Gamma, \tau) \\
 & & \\
[\![\Gamma \vdash K\, e_1 \ldots e_n\colon \tau]\!] & = & \exists \bar{\alpha}'.(\wedge_i [\![\Gamma \vdash e_i\colon \tau_i[\bar{\alpha} := \bar{\alpha}']]\!] \wedge \varepsilon(\bar{\alpha}') \doteq \tau), \\
 & & \text{w. } K :: \forall \bar{\alpha}.\tau_1 \times \ldots \times \tau_n \to \varepsilon(\bar{\alpha}), \bar{\alpha}' \# \mathrm{FV}(\Gamma, \tau) \\
 & & \\
[\![\Gamma \vdash e\colon \tau]\!] & = & (\exists \beta.C) \wedge [\![\Gamma\{x\colon \forall \beta[C].\beta\} \vdash e_2\colon \tau]\!] \\
e = \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 & & \text{where } C = [\![\Gamma \vdash e_1\colon \beta]\!] \\
 & & \\
[\![\Gamma \vdash e\colon \tau]\!] & = & (\exists \beta.C) \wedge [\![\Gamma\{x\colon \forall \beta[C].\beta\} \vdash e_2\colon \tau]\!] \\
e = \mathbf{letrec}\ x = e_1\ \mathbf{in}\ e_2 & & \text{where } C = [\![\Gamma\{x\colon \beta\} \vdash e_1\colon \beta]\!]
\end{array}
$$

$$
\begin{array}{rcl}
[\![\Gamma \vdash e\colon \tau]\!] & = & \exists \alpha_v.[\![\Gamma \vdash e_v\colon \alpha_v]\!] \wedge_i [\![\Gamma \vdash p_i.e_i\colon \alpha_v \to \tau]\!], \\
e = \mathbf{match}\ e_v\ \mathbf{with}\ \bar{c} & & \alpha_v \# \mathrm{FV}(\Gamma, \tau) \\
\bar{c} = p_1.e_1|...|p_n.e_n & &
\end{array}
$$

$$
[\![\Gamma, \Sigma \vdash p.e\colon \tau_1 \to \tau_2]\!] = [\![\Sigma \vdash p{\downarrow}\tau_1]\!] \wedge \exists \bar{\beta}.[\![\Gamma\Gamma' \vdash e\colon \tau_2]\!]
$$

where $\exists \bar{\beta}\Gamma'$ is $[\![\Sigma \vdash p{\uparrow}\tau_1]\!], \bar{\beta} \# \mathrm{FV}(\Gamma, \tau_2)$

$[\![\Sigma \vdash p{\downarrow}\tau_1]\!]$ derives constraints on type of matched value

$[\![\Sigma \vdash p{\uparrow}\tau_1]\!]$ derives environment for pattern variables

- By $\bar{\alpha}$ or $\bar{\alpha_i}$ we denote a sequence of some length: $\alpha_1...\alpha_n$
- By $\wedge_i \varphi_i$ we denote a conjunction of $\bar{\varphi_i}$: $\varphi_1...\varphi_n$.

Polymorphic Recursion

- Note the limited polymorphism of `let rec f = ...` – we cannot use `f` polymorphically in its definition.
  - In modern OCaml we can bypass the problem if we provide type of `f` upfront: `let rec f : 'a. 'a -> 'a list = ...`
  - where `'a. 'a -> 'a list` stands for $\forall\alpha.\alpha \to \alpha\,\mathrm{list}$.
- Using the recursively defined function with different types in its definition is called polymorphic recursion.
- It is most useful together with irregular recursive datatypes where the recursive use has different type arguments than the actual parameters.

Polymorphic Rec: A list alternating between two types of elements

```
type ('x, 'o) alterning =
| Stop
| One of 'x * ('o, 'x) alterning

let rec to_list :
    'x 'o 'a. ('x->'a) -> ('o->'a) ->
            ('x, 'o) alterning -> 'a list =
  fun x2a o2a ->
    function
    | Stop -> []
    | One (x, rest) -> x2a x::to_list o2a x2a rest

let to_choice_list alt =
  to_list (fun x->Left x) (fun o->Right o) alt

let it = to_choice_list
  (One (1, One ("o", One (2, One ("oo", Stop)))))
```

Polymorphic Rec: Data-Structural Bootstrapping

```
type 'a seq = Nil | Zero of ('a * 'a) seq | One of 'a * ('a * 'a) seq
```

We store a list of elements in exponentially increasing chunks.

```
let example =
  One (0, One ((1,2), Zero (One ((((3,4),(5,6)), ((7,8),(9,10))), Nil))))

let rec cons : 'a. 'a -> 'a seq -> 'a seq =
  fun x -> function
  | Nil -> One (x, Nil)
  | Zero ps -> One (x, ps)
  | One (y, ps) -> Zero (cons (x,y) ps)
```

Appending an element to the datastructure is like adding one to a binary number: 1+0=1, 1+...0=...1, 1+...1=[...+1]0

```
let rec lookup : 'a. int -> 'a seq -> 'a =
  fun i s -> match i, s with
  | _, Nil -> raise Not_found
  | 0, One (x, _) -> x
  | i, One (_, ps) -> lookup (i-1) (Zero ps)
  | i, Zero ps ->
    let x, y = lookup (i / 2) ps in
    if i mod 2 = 0 then x else y
```

Rather than returning `None : 'a option` we raise exception, for convenience.

Random-Access lookup works in logarithmic time – much faster than in standard lists.

Algebraic Specification

- The way we introduce a data structure, like complex numbers or strings, in mathematics, is by specifying an *algebraic structure*.
- Algebraic structures consist of a set (or several sets, for so-called *multi-sorted* algebras) and a bunch of functions (aka. operations) over this set (or sets).
- A *signature* is a rough description of an algebraic structure: it provides sorts – names for the sets (in multisorted case) and names of the functions-operations together with their arity (and what sorts of arguments they take).
- We select a class of algebraic structures by providing axioms that have to hold. We will call such classes *algebraic specifications*.
  - In mathematics, a rusty name for some algebraic specifications is a *variety*, a more modern and name is *algebraic category*.
- Algebraic structures correspond to "implementations" and signatures to "interfaces" in programming languages.

- We will say that an algebraic structure implements an algebraic specification when all axioms of the specification hold in the structure.
- All algebraic specifications are implemented by multiple structures!
- We say that an algebraic structure does not have junk, when all its elements (i.e. elements in the sets corresponding to sorts) can be built using operations in its signature.
- We allow parametric types as sorts. In that case, strictly speaking, we define a family of algebraic specifications (a different specification for each instantiation of the parametric type).

Algebraic specifications: examples

- An algebraic specification can also use an earlier specification.
- In "impure" languages like OCaml and F# we allow that the result of any operation be an error. In Haskell we could use `Maybe`.

nat_p
$0\colon \mathrm{nat}_p$ $\mathrm{succ}\colon \mathrm{nat}_p \to \mathrm{nat}_p$ $+\colon \mathrm{nat}_p \to \mathrm{nat}_p \to \mathrm{nat}_p$ $*\colon \mathrm{nat}_p \to \mathrm{nat}_p \to \mathrm{nat}_p$
$n, m\colon \mathrm{nat}_p$
$0 + n = n,\ n + 0 = n$ $m + \mathrm{succ}(n) = \mathrm{succ}(m + n)$ $0 * n = 0,\ n * 0 = 0$ $m * \mathrm{succ}(n) = m + (m * n)$ $\mathrm{succ}(\ldots\mathrm{succ}(0)) \neq 0$ less than p times $\mathrm{succ}(\ldots\mathrm{succ}(0)) = 0$ p times

string_p
uses char, nat_p
$"" \colon \mathrm{string}_p$ $"\cdot" \colon \mathrm{char} \to \mathrm{string}_p$ $\hat{}\colon \mathrm{string}_p \to \mathrm{string}_p \to \mathrm{string}_p$ $\cdot[\cdot]\colon \mathrm{string}_p \to \mathrm{nat}_p \to \mathrm{char}$
$s\colon \mathrm{string}_p, c, c_1, \ldots, c_p\colon \mathrm{char}, n\colon \mathrm{nat}_p$
$"" \hat{} s = s,\ s \hat{} "" = s$ $"c_1" \hat{} (\ldots \hat{} "c_p") = \mathrm{error}$ p times $r \hat{} (s \hat{} t) = (r \hat{} s) \hat{} t$ $("c" \hat{} s)[0] = c$ $("c" \hat{} s)[\mathrm{succ}(n)] = s[n]$ $""[n] = \mathrm{error}$

Homomorphisms

- Mappings between algebraic structures with the same signature preserving operations.
- A *homomorphism* from algebraic structure $(A, \{f^A, g^A, ...\})$ to $(B, \{f^B, g^B, ...\})$ is a function $h\colon A \to B$ such that $h(f^A(a_1, ..., a_{n_f})) = f^B(h(a_1), ..., h(a_{n_f}))$ for all $(a_1, ..., a_{n_f})$, $h(g^A(a_1, ..., a_{n_g})) = g^B(h(a_1), ..., h(a_{n_g}))$ for all $(a_1, ..., a_{n_g})$, ...
- Two algebraic structures are *isomorphic* if there are homomorphisms $h_1\colon A \to B$, $h_2\colon B \to A$ from one to the other and back, that when composed in any order form identity: $\forall(b \in B)\, h_1(h_2(b)) = b$, $\forall(a \in A)\, h_2(h_1(a)) = a$.
- An algebraic specification whose all implementations without junk are isomorphic is called "*monomorphic*".
  - We usually only add axioms that really matter to us to the specification, so that the implementations have room for optimization. For this reason, the resulting specifications will often not be monomorphic in the above sense.

Example: Maps

$(\alpha, \beta)\,\mathrm{map}$, or $\mathrm{map}\langle\alpha, \beta\rangle$
uses bool, type parameters α, β
$\mathrm{empty}\colon (\alpha, \beta)\,\mathrm{map}$ $\mathrm{member}\colon \alpha \to (\alpha, \beta)\,\mathrm{map} \to \mathrm{bool}$ $\mathrm{add}\colon \alpha \to \beta \to (\alpha, \beta)\,\mathrm{map} \to (\alpha, \beta)\,\mathrm{map}$ $\mathrm{remove}\colon \alpha \to (\alpha, \beta)\,\mathrm{map} \to (\alpha, \beta)\,\mathrm{map}$ $\mathrm{find}\colon \alpha \to (\alpha, \beta)\,\mathrm{map} \to \beta$
$k, k_2\colon \alpha,\ v, v_2\colon \beta,\ m\colon (\alpha, \beta)\,\mathrm{map}$
$\mathrm{member}(k, \mathrm{add}(k, v, m)) = \mathrm{true}$ $\mathrm{member}(k, \mathrm{remove}(k, m)) = \mathrm{false}$ $\mathrm{member}(k, \mathrm{add}(k_2, v, m)) = \mathrm{true} \wedge k \neq k_2 \Leftrightarrow \mathrm{member}(k, m) = \mathrm{true} \wedge k \neq k_2$ $\mathrm{member}(k, \mathrm{remove}(k_2, v, m)) = \mathrm{true} \wedge k \neq k_2 \Leftrightarrow \mathrm{member}(k, m) = \mathrm{true} \wedge k \neq k_2$ $\mathrm{find}(k, \mathrm{add}(k, v, m)) = v$ $\mathrm{find}(k, \mathrm{remove}(k, m)) = \mathrm{error},\ \mathrm{find}(k, \mathrm{empty}) = \mathrm{error}$ $\mathrm{find}(k, \mathrm{add}(k_2, v_2, m)) = v \wedge k \neq k_2 \Leftrightarrow \mathrm{find}(k, m) = v \wedge k \neq k_2$ $\mathrm{find}(k, \mathrm{remove}(k_2, v_2, m)) = v \wedge k \neq k_2 \Leftrightarrow \mathrm{find}(k, m) = v \wedge k \neq k_2$ $\mathrm{remove}(k, \mathrm{empty}) = \mathrm{empty}$

Modules and interfaces (signatures): syntax

- In the ML family of languages, structures are given names by **module** bindings, and signatures are types of modules.
- From outside of a structure or signature, we refer to the values or types it provides with a dot notation: `Module.value`.
- Module (and module type) names have to start with a capital letter (in ML languages).
- Since modules and module types have names, there is a tradition to name the central type of a signature (the one that is "specified" by the signature), for brevity, `t`.
- Module types are often named with "all-caps" (all letters upper case).

```
module type MAP = sig
  type ('a, 'b) t
  val empty : ('a, 'b) t
  val member : 'a -> ('a, 'b) t -> bool
  val add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t
  val remove : 'a -> ('a, 'b) t -> ('a, 'b) t
  val find : 'a -> ('a, 'b) t -> 'b
end

module ListMap : MAP = struct
  type ('a, 'b) t = ('a * 'b) list
  let empty = []
  let member = List.mem_assoc
  let add k v m = (k, v)::m
  let remove = List.remove_assoc
  let find = List.assoc
end
```

Implementing maps: Association lists

Let's now build an implementation of maps from the ground up. The most straightforward implementation... might not be what you expected:

```
module TrivialMap : MAP = struct
  type ('a, 'b) t =
    | Empty
    | Add of 'a * 'b * ('a, 'b) t
    | Remove of 'a * ('a, 'b) t
  let empty = Empty
  let rec member k m =
    match m with
      | Empty -> false
      | Add (k2, _, _) when k = k2 -> true
      | Remove (k2, _) when k = k2 -> false
      | Add (_, _, m2) -> member k m2
      | Remove (_, m2) -> member k m2
  let add k v m = Add (k, v, m)
  let remove k m = Remove (k, m)
```

```
  let rec find k m =
    match m with
      | Empty -> raise Not_found
      | Add (k2, v, _) when k = k2 -> v
      | Remove (k2, _) when k = k2 -> raise Not_found
      | Add (_, _, m2) -> find k m2
      | Remove (_, m2) -> find k m2
end
```

Here is an implementation based on association lists, i.e. on lists of key-value pairs.

```
module MyListMap : MAP = struct
  type ('a, 'b) t = Empty | Add of 'a * 'b * ('a, 'b) t
  let empty = Empty
  let rec member k m =
    match m with
      | Empty -> false
      | Add (k2, _, _) when k = k2 -> true
      | Add (_, _, m2) -> member k m2
  let rec add k v m =
    match m with
      | Empty -> Add (k, v, Empty)
      | Add (k2, _, m) when k = k2 -> Add (k, v, m)
      | Add (k2, v2, m) -> Add (k2, v2, add k v m)
```

```
  let rec remove k m =
    match m with
      | Empty -> Empty
      | Add (k2, _, m) when k = k2 -> m
      | Add (k2, v, m) -> Add (k2, v, remove k m)
  let rec find k m =
    match m with
      | Empty -> raise Error
      | Add (k2, v, _) when k = k2 -> v
      | Add (_, _, m2) -> find k m2
end
```

Implementing maps: Binary search trees

- Binary search trees are binary trees with elements stored at the interior nodes, such that elements to the left of a node are smaller than, and elements to the right bigger than, elements within a node.
- For maps, we store key-value pairs as elements in binary search trees, and compare the elements by keys alone.
- On average, binary search trees are fast because they use "divide-and-conquer" to search for the value associated with a key. ($O(\log n)$ compl.)
  - In worst case they reduce to association lists.
- The simple polymorphic signature for maps is only possible with implementations based on some total order of keys because OCaml has polymorphic comparison (and equality) operators.
  - These operators work on elements of most types, but not on functions. They may not work in a way you would want though!
  - Our signature for polymorphic maps is not the standard approach because of the problem of needing the order of keys; it is just to keep things simple.

```
module BTreeMap : MAP = struct
  type ('a, 'b) t = Empty | T of ('a, 'b) t * 'a * 'b * ('a, 'b) t
  let empty = Empty
  let rec member k m =
    match m with
      | Empty -> false
      | T (_, k2, _, _) when k = k2 -> true
      | T (m1, k2, _, _) when k < k2 -> member k m1
      | T (_, _, _, m2) -> member k m2
  let rec add k v m =
    match m with
      | Empty -> T (Empty, k, v, Empty)
      | T (m1, k2, _, m2) when k = k2 -> T (m1, k, v, m2)
      | T (m1, k2, v2, m2) when k < k2 -> T (add k v m1, k2, v2, m2)
      | T (m1, k2, v2, m2) -> T (m1, k2, v2, add k v m2)
  let rec split_rightmost m =
    match m with
      | Empty -> raise Not_found
      | T (Empty, k, v, Empty) -> k, v, Empty
      | T (m1, k, v, m2) ->
        let rk, rv, rm = split_rightmost m2 in
        rk, rv, T (m1, k, v, rm)
```

"Divide and conquer" search through the tree.

Searches the tree in the same way as `member` but copies every node along the way.

A helper function, it does not belong to the "exported" signature.

We remove one element, the one that is on the bottom right.

```
  let rec remove k m =
    match m with
      | Empty -> Empty
      | T (m1, k2, _, Empty) when k = k2 -> m1
      | T (Empty, k2, _, m2) when k = k2 -> m2
      | T (m1, k2, _, m2) when k = k2 ->
        let rk, rv, rm = split_rightmost m1 in
        T (rm, rk, rv, m2)
      | T (m1, k2, v, m2) when k < k2 -> T (remove k m1, k2, v, m2)
      | T (m1, k2, v, m2) -> T (m1, k2, v, remove k m2)
  let rec find k m =
    match m with
      | Empty -> raise Not_found
      | T (_, k2, v, _) when k = k2 -> v
      | T (m1, k2, _, _) when k < k2 -> find k m1
      | T (_, _, _, m2) -> find k m2
end
```

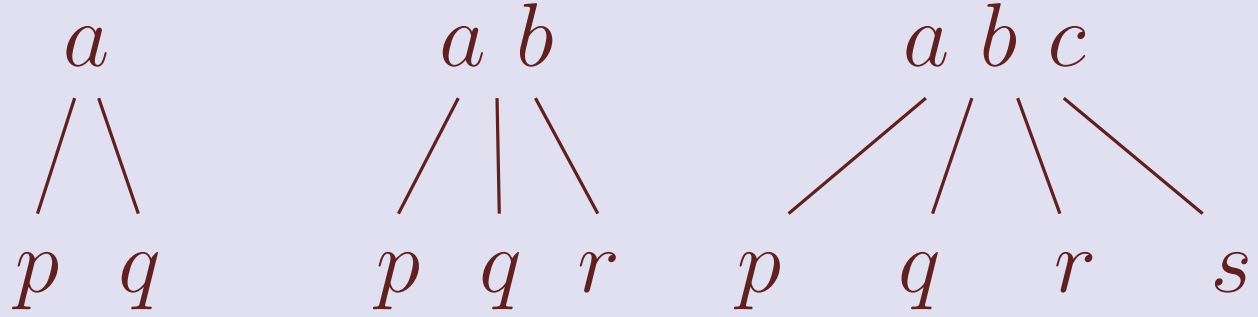
Implementing maps: red-black trees

Based on Wikipedia http://en.wikipedia.org/wiki/Red-black_tree, Chris Okasaki's "Functional Data Structures" and Matt Might's excellent blog post http://matt.might.net/articles/red-black-delete/.

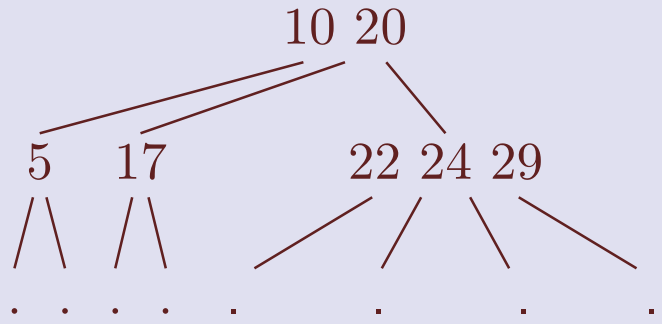
- Binary search trees are good when we encounter keys in random order, because the cost of operations is limited by the depth of the tree which is small relatively to the number of nodes...
- ...unless the tree grows unbalanced achieving large depth (which means there are sibling subtrees of vastly different sizes on some path).
- To remedy it, we rebalance the tree while building it – i.e. while adding elements.
- In *red-black trees* we achieve balance by remembering one of two colors with each node, keeping the same length of each root-leaf path if only black nodes are counted, and not allowing a red node to have a red child.
  - This way the depth is at most twice the depth of a perfectly balanced tree with the same number of nodes.

B-trees of order 4 (2-3-4 trees)

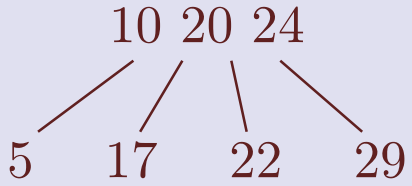
How can we have perfectly balanced trees without worrying about having $2^k - 1$ elements? **2-3-4 trees** can store from 1 to 3 elements in each node and have 2 to 4 subtrees correspondingly. Lots of freedom!



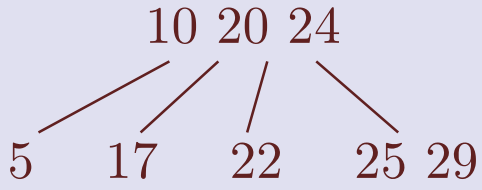
To insert “25” into (“.” stand for leaves, ignored later)



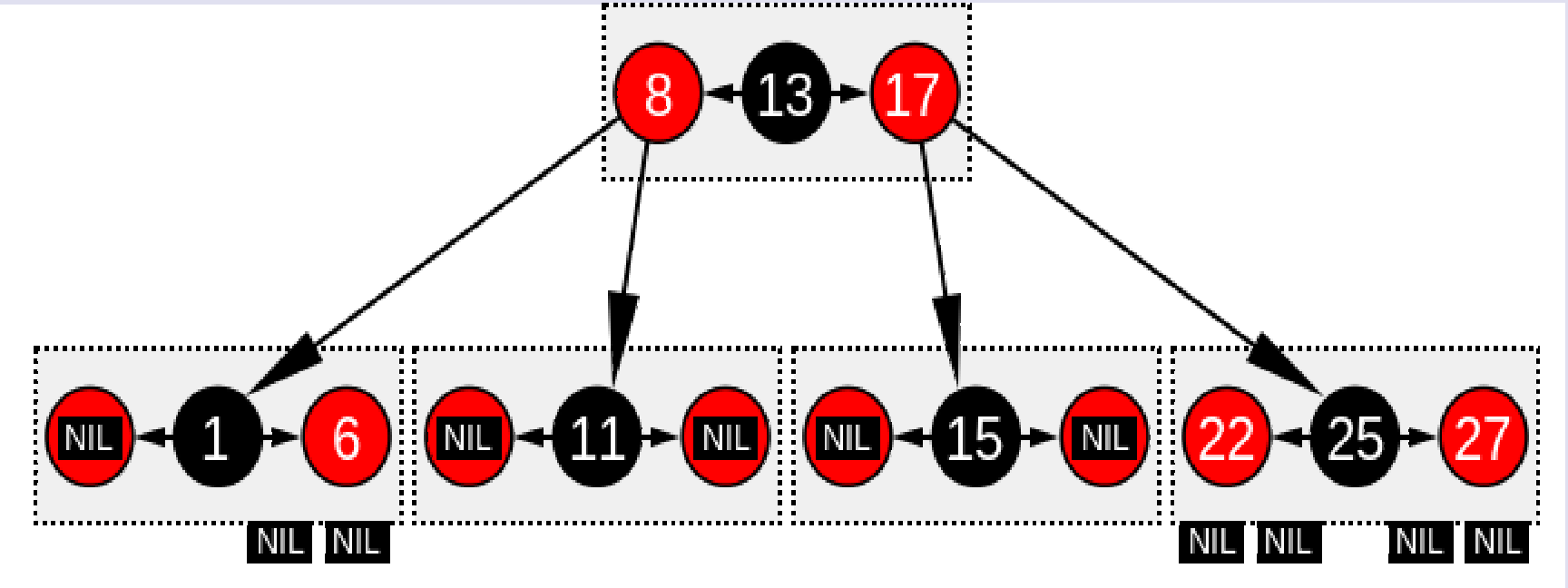
we descend right, but it is a full node, so we move the middle up and split the remaining elements:



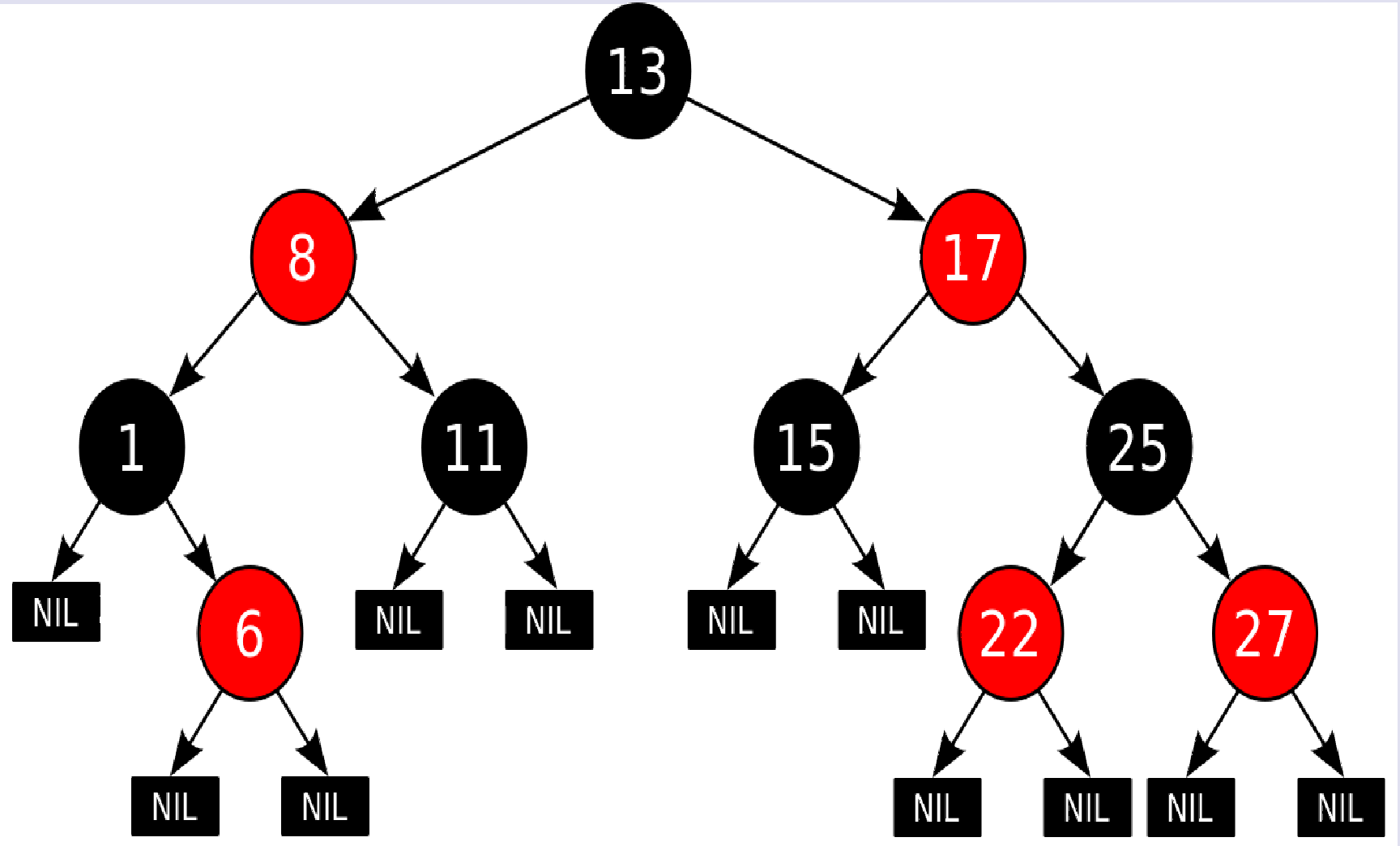
Now there is a place between 24 and 29: next to 29



To represent 2-3-4 tree as a binary tree with one element per node, we color the middle element of a 4-node, or the first element of 2-/3-node, black and make it the parent of its neighbor elements, and make them parents of the original subtrees. Turning this:



into this Red-Black tree:



Red-Black trees, without deletion

- **Invariant 1.** No red node has a red child.
- **Invariant 2.** Every path from the root to an empty node contains the same number of black nodes.
- First we implement Red-Black tree based sets without deletion.
- The implementation proceeds almost exactly like for unbalanced binary search trees, we only need to restore invariants.
- By keeping balance at each step of constructing a node, it is enough to check locally (around the root of the subtree).
- For understandable implementation of deletion, we need to introduce more colors. See Matt Might's post edited in a separate file.

```
type color = R | B
type 'a t = E | T of color * 'a t * 'a * 'a t
let empty = E
let rec member x m =
  match m with
  | Empty -> false
  | T (_, _, y, _) when x = y -> true
  | T (_, a, y, _) when x < y -> member x a
  | T (_, _, _, b) -> member x b
let balance = function
  | B,T (R,T (R,a,x,b),y,c),z,d
  | B,T (R,a,x,T (R,b,y,c)),z,d
  | B,a,x,T (R,T (R,b,y,c),z,d)
  | B,a,x,T (R,b,y,T (R,c,z,d))
    -> T (R,T (B,a,x,b),y,T (B,c,z,d))
  | color,a,x,b -> T (color,a,x,b)
```

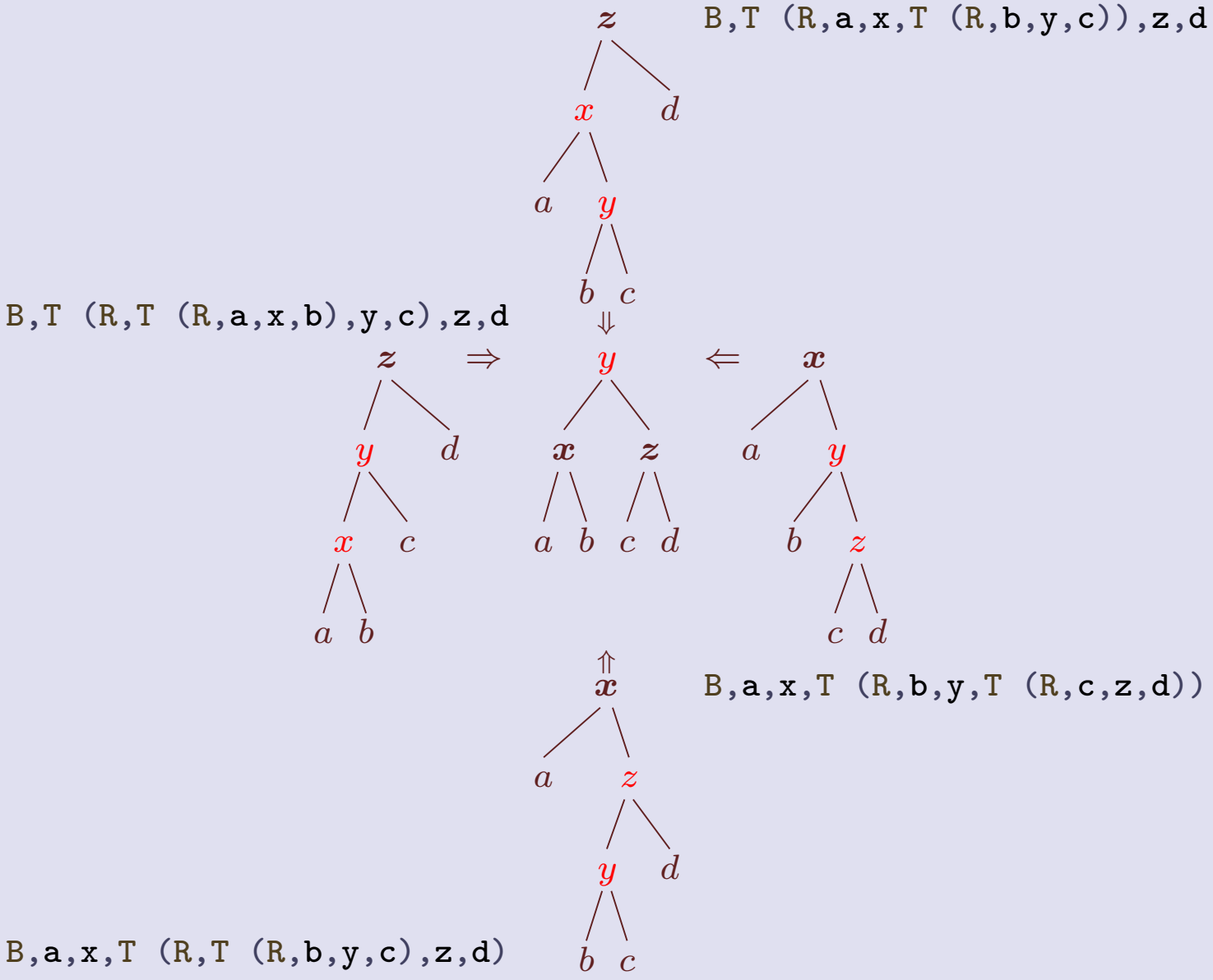
Like in unbalanced binary search tree.

Restoring the invariants.
On next figure: left,
top,
bottom,
right,
center tree.
We allow red-red violation for now.

```
let insert x s =
  let rec ins = function
    | E -> T (R,E,x,E)
    | T (color,a,y,b) as s ->
      if x<y then balance (color,ins a,y,b)
      else if x>y then balance (color,a,y,ins b)
      else s in
  match ins s with
  | T (_,a,y,b) -> T (B,a,y,b)
  | E -> failwith "insert: impossible"
```

Like in unbalanced binary search tree, but fix violation above created node.

We could still have red-red violation at root, fixed by coloring it black.


B,T (R,a,x,T (R,b,y,c)),z,d
B,T (R,T (R,a,x,b),y,c),z,d
B,a,x,T (R,b,y,T (R,c,z,d))
B,a,x,T (R,T (R,b,y,c),z,d)

Homework

1. Derive the equations and solve them to find the type for:

   ```
   let cadr l = List.hd (List.tl l) in
   cadr (1::2::[]), cadr (true::false::[])
   ```

   in environ. $\Gamma = \{$`List.hd`$: \forall\alpha.\alpha\ \text{list} \to \alpha;$ `List.tl`$: \forall\alpha.\alpha\ \text{list} \to \alpha\ \text{list}\}$. You can take "shortcuts" if it is too many equations to write down.

2. What does it mean that an implementation has junk (as an algebraic structure for a given signature)? Is it bad?

3. Define a monomorphic algebraic specification (other than, but similar to, nat_p or string_p, some useful data type).

4. Discuss an example of a (monomorphic) algebraic specification where it would be useful to drop some axioms (giving up monomorphicity) to allow more efficient implementations.

5. Does the example `ListMap` meet the requirements of the algebraic specification for maps? Hint: here is the definition of `List.remove_assoc`; `compare a x` equals `0` if and only if `a = x`.

```
let rec remove_assoc x = function
  | [] -> []
  | (a, b as pair) :: l ->
      if compare a x = 0 then l else pair :: remove_assoc x l
```

6. Trick question: what is the computational complexity of `ListMap` or `TrivialMap`?

7. * The implementation `MyListMap` is inefficient: it performs a lot of copying and is not tail-recursive. Optimize it (without changing the type definition).

8. Add (and specify) $\mathrm{isEmpty}\colon (\alpha, \beta)\ \mathrm{map} \to \mathrm{bool}$ to the example algebraic specification of maps without increasing the burden on its implementations (i.e. without affecting implementations of other operations). Hint: equational reasoning might be not enough; consider an equivalence relation $\approx$ meaning "have the same keys", defined and used just in the axioms of the specification.

9. Design an algebraic specification and write a signature for first-in-first-out queues. Provide two implementations: one straightforward using a list, and another one using two lists: one for freshly added elements providing efficient queueing of new elements, and "reversed" one for efficient popping of old elements.

10. Design an algebraic specification and write a signature for sets. Provide two implementations: one straightforward using a list, and another one using a map into the unit type.

11. (Ex. 2.2 in Chris Okasaki "Purely Functional Data Structures") In the worst case, `member` performs approximately $2d$ comparisons, where d is the depth of the tree. Rewrite `member` to take no mare than $d + 1$ comparisons by keeping track of a candidate element that *might* be equal to the query element (say, the last element for which $<$ returned false) and checking for equality only when you hit the bottom of the tree.

12. (Ex. 3.10 in Chris Okasaki "Purely Functional Data Structures") The `balance` function currently performs several unnecessary tests: when e.g. `ins` recurses on the left child, there are no violations on the right child.

    a. Split `balance` into `lbalance` and `rbalance` that test for violations of left resp. right child only. Replace calls to `balance` appropriately.

    b. One of the remaining tests on grandchildren is also unnecessary. Rewrite `ins` so that it never tests the color of nodes not on the search path.

13. * Implement maps (i.e. write a module for the map signature) based on AVL trees. See `http://en.wikipedia.org/wiki/AVL_tree`.