FDD3024 Lecture 3: Data types, polymorphism, abstract types, module systems

Karl Palmskog

KTH

Monday April 20, 2026

Course material

- PFPL chapter 9, natural numbers
- PFPL chapters 10 and 11, finite data types
- PFPL Chapter 16, polymorphic types and functions
- PFPL chapter 17, abstract types
- PFPL chapter 44, modules and type classes

Why (not) data types?

- data can be encoded as functions, so not strictly necessary
- but convenient in many applications
- progress/preservation holds once and for all
- domain modeling tool

Church numerals

- 0: $\lambda f.\lambda x.x$
- 1: $\lambda f.\lambda x.fx$
- 2: $\lambda f.\lambda x.f(fx)$
- 3: $\lambda f.\lambda x.f(f(fx))$
- successor: $\lambda n.\lambda f.\lambda x.f(n\, f\, x)$

Natural numbers and recursors

t	$::=$			term
	$\mid$	x		variable
	$\mid$	$\lambda x.t$	bind x in t	lambda
	$\mid$	$t\,t'$		app
	$\mid$	$\mathbf{z}$		zero
	$\mid$	$\mathbf{s}\,(t)$		successor
	$\mid$	$\mathbf{rec}\,(t, t_0, x.y.t_1)$		recursion
	$\mid$	(t)	S	
	$\mid$	$[t/x]t'$	M	
$typ,\ T$	$::=$			types
	$\mid$	Nat		natural numbers
	$\mid$	$T_1 \to T_2$		function types

Typing numbers and recursors

$$\frac{}{\Gamma \vdash \mathbf{z} : \mathsf{Nat}} \ \textsc{typing\_z}$$

$$\frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathbf{s}\,(t) : \mathsf{Nat}} \ \textsc{typing\_s}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \mathsf{Nat} \\ \Gamma \vdash t_0 : T \\ \Gamma \vdash x : \mathsf{Nat} \\ \Gamma \vdash y : T \\ \Gamma \vdash t_1 : T\end{array}}{\Gamma \vdash \mathbf{rec}\,(t, t_0, x.y.t_1) : T} \ \textsc{typing\_rec}$$

Products

- mathematical “tuples” at programming language level
- basic form of structured data
- ubiquitous

Binary product syntax

$$
\begin{array}{lll}
T & ::= & \\
 & | & \mathsf{unit} \\
 & | & \mathsf{prod}\,(T_1, T_2) \\
 & & \\
e & ::= & \\
 & | & \mathbf{triv} \\
 & | & \mathsf{pair}\,(e_1, e_2) \\
 & | & \mathsf{pr}\,[\,\mathbf{l}\,](e) \\
 & | & \mathsf{pr}\,[\,\mathbf{r}\,](e)
\end{array}
$$

General product syntax

$$
\begin{array}{lll}
T & ::= & \\
 & | & i_1 \rightsquigarrow T_1 * \ldots * i_n \rightsquigarrow T_n \\
 & & \\
e & ::= & \\
 & | & < i_1 \rightsquigarrow e_1, \ldots, i_n \rightsquigarrow e_n > \\
 & | & \mathsf{pr}\,[i](e)
\end{array}
$$

Sum types

- representation of finite choices
- choice determines data structure
- leaf vs. branch, something vs. nothing

Sum syntax

$$
\begin{array}{lll}
T & ::= & \\
 & | & \mathsf{void} \\
 & | & \mathsf{sum}\,(T_1, T_2) \\
 & & \\
e & ::= & \\
 & | & \mathsf{case}\,\{T\}(e) \\
 & | & \mathsf{in}\,[\mathsf{l}]\{T_1, T_2\}(e) \\
 & | & \mathsf{in}\,[\mathsf{r}]\{T_1, T_2\}(e) \\
 & | & \mathsf{case}\,(e, x_1.e_1, x_2.e_2)
\end{array}
$$

Sum type applications

- assume we have `unit` type
- boolean type can then be encoded as sum$(\text{unit}, \text{unit})$
- if statement becomes `case` expression
- works for any "enumeration" type, like suits in cards
- option type is sum(unit, T) for some type T
  - option type value either has T value or doesn't

Parametric polymorphism

- lists and other "parameterized" data types are obiquitous
- we want to define functions and prove properties once and for all parameters
  - avoid "redundant code"
- this is usually called *parametric* polymorphism in contrast to "ad-hoc polymorphism"
- parametric polymorphism is typically via type variables
  - Girard's System F
  - Reynolds' polymorphic typed λ-calculus
  - related to Java generics and C++ templates
- see PFPL chapter 16 for a more formal account

Parametric polymorphism for lists

```
let rec f (p: 'a -> bool) (l:list 'a) : bool =
 match l with
 | [] -> true
 | x::r -> p x && f p r
 end
```

- we can "instantiate" `'a` for any type
- correctness proofs can be done implicitly quantified over all `'a`
- type checking can be done for polymorphic code with `'a`, `'b`, etc.
- substituting concrete types for all `'a`, `'b`, etc., yields well typed expressions

Data abstraction

- interfaces are a form of agreement between implementer and client of a program
- interfaces should isolate the client from the implementer
- a compliant implementation should be replacable by another by the client without affecting (functional) behavior

Abstract types

- abstract types are **existential** types, we have no knowledge about their representation
- we define collections of operations on the unknown type
- two mechanisms:
  - information hiding (for client)
  - compliance checking (for implementer)
- also facilitates separate compilation

Harper's queue of natural numbers

Consider an abstract type of FIFO queues supporting three operations:

- forming the empty queue (`emp`)
- inserting a natural number at the end of the queue (`ins`)
- removing the natural number at the head of the queue (`rem`)

Type signatures

- by convention, the existential type is called `t`
- form empty: `emp : t`
- insertion: `ins : nat*t -> t`
- removal: `rem : t -> (nat*t) option`
- the existential type is formed by product of all operation types

Packing and opening existential types

- expressions of type $\exists(t.\tau)$ are "packages" of form
  pack ρ with e as $\exists(t.\tau)$
  - ρ is a type ("representation type")
  - e is expression of type $[\rho/t]\tau$ ("implementation")
- to use a package, use elimination form
  open e_1 as t with $x : \tau$ in e_2
  - e_1 is expression of type $\exists(t.\tau)$
  - e_2 is the "client expression" with some type τ_2, may implement a sequence of operations via x

Existential types in CakeML

- declare a `structure` (namespace)
- open a `local .. in ..` block inside `structure`
- first declare type and all operations, including auxiliary functions
- then declare interface operations
- implementation (including type) is hidden, but swapping implementations cumbersome
- see example code in supplementary material
- need full module system (signatures) for convenient implementation swapping

Existential types in OCaml

```
type 'a t
(* The type of stacks containing elements of type ['a]. *)

val create : 'a t
(* Return an empty stack. *)

val push : 'a -> 'a t -> 'a t
(* [push x s] returns the stack that has [x] at the
 top of stack [s]. *)

val pop_opt : 'a t -> ('a * 'a t) option
(* [pop_opt s] returns the topmost element in
   [s] and [s] with element removed, or [None]
   if stack is empty. *)
```

Representation independence

- should be possible to ensure clients are unaffected by swapping implementations of the same abstract type
- Harper proposes concept of **bisimilarity** to formalize what "unaffected" means
- informally, implementations are bisimilar when observers (clients) can't tell them apart by interacting with them
- ignoring running time, resource consumption, etc.

Bisimulation proof method

- to prove correctness of a candidate implementation of abstract type, show that it is bisimilar to an obviously correct reference implementation
- similar to using a "functional model" in contracts
- if proof succeeds, no client can distinguish if they are using reference implementation or candidate
  - typically assume resource use (time, memory, etc.) is not observable
  - security properties also may not be preserved

Bisimulation proof setup

- reference implementation of queue (e.g., using single list):
  - emp: e_m
  - ins: e_i
  - del: e_r
- candidate implementation of queue (e.g., using two lists):
  - emp: e'_m
  - ins: e'_i
  - del: e'_r
- find binary relation R between expressions from reference and candidate implementations
  - empty queues should be related
  - inserting same element into related queues should yield related queues
  - deleting same element from related queues should yield related expressions

Modules

- data and functions are programming in the small
  - small pieces of functionality
  - small building blocks of data
- modules (and type classes) are about programming in the large
  - “separable and reusable components”
  - large utility libraries intended for reuse
  - large-scale software system construction

Example: SML queue module type

```
signature QUEUE = sig
 type 'a Queue
 val empty : 'a Queue
 val isEmpty : 'a Queue -> bool
 val snoc : 'a Queue * 'a -> 'a Queue
 val head : 'a Queue -> 'a
 val tail : 'a Queue -> 'a Queue
end
```

Example: SML ordered module type

```
signature ORDERED = sig
 type T
 val eq : T * T -> bool
 val lt : T * T -> bool
 val leq : T * T -> bool
end

structure IntOrd : ORDERED = struct
 type T = int
 val eq = (=)
 val lt = (<)
 val leq = (<=)
end
```

Example: SML sortable signature

```
signature SORTABLE = sig
  structure Elem : ORDERED
  type Sortable
  val empty : Sortable
  val add : Elem.T * Sortable -> Sortable
  val sort : Sortable -> Elem.T list
end
```

Example: Haskell sortable type class

```
class Sortable s where
 empty :: s a
 add :: Ord a => a -> s a -> s a
 sort :: Ord a => s a -> [a]

instance Sortable T.RBTree where
 empty = T.empty
 add = T.insert
 sort = T.toList
```

Example: SML heap module type

```
signature HEAP = sig
 structure Elem : ORDERED
 type Heap
 val empty : Heap
 val isEmpty : Heap -> bool
 val insert : Elem.T * Heap -> Heap
 val merge : Heap * Heap -> Heap
 val findMin : Heap -> Elem.T
 val deleteMin : Heap -> Heap
end
```

Example: Haskell heap type class

```
class Heap h where
 empty     :: Ord a => h a
 isEmpty   :: Ord a => h a -> Bool
 insert    :: Ord a => a -> h a -> h a
 merge     :: Ord a => h a -> h a -> h a
 findMin   :: Ord a => h a -> a
 deleteMin :: Ord a => h a -> h a
```

Example: SML functor

```
functor SizedHeap (H : HEAP) : HEAP = struct
 structure Elem = H.Elem
 datatype Heap = NE of int * H.Heap
 val empty = NE (0, H.empty)
 fun isEmpty NE (n, h) = (n = 0)
 fun insert (x, NE (n, h)) =
  NE (n + 1, H.insert (x, h))
 fun merge (NE (n1, h1), NE (n2, h2)) =
  NE (n1 + n2, H.merge (h1, h2))
 fun findMin NE (n, h) = H.findMin h
 fun deleteMin NE (n, h) =
  NE (n - 1, H.deleteMin h)
end
```

Example : SML functor

```
functor QueueWithCons (Q : QUEUE) : QUEUE = struct
 type 'a Queue = 'a list * 'a Q.Queue
 val empty = ([], Q.empty)
 fun isEmpty ([], q) = Q.isEmpty q | isEmpty _ = false
 fun cons (x, (xs, q)) = (x::xs, q)
 fun snoc ((xs, q), x) = (h, Q.snoc (q, x))
 fun head ([], q) = Q.head q
   | head (x::xs, q) = x
 fun tail ([], q) = Q.tail q
   | tail (x::xs, q) = (xs, q)
end
```

A. Rossberg on SML modules

"ML is two languages in one: there is the core, with types and expressions, and there are modules, with signatures, structures and functors."

"Modules form a separate, higher-order functional language on top of the core. There are both practical and technical reasons for this stratification; yet, it creates substantial duplication in syntax and semantics, and it reduces expressiveness. For example, selecting a module cannot be made a dynamic decision."

https://people.mpi-sws.org/~rossberg/1ml/

First- and second-class modules

- are module definitions just expressions?
- first-class module values can depend on runtime
  - can be convenient when passing options via command line
  - chosen for OCaml
- second-class module values are statically determined
  - reasoning and type checking much easier
  - chosen for Standard ML

Module-based libraries in production

- Standard ML Basis library (pioneer)
- OCaml Stdlib
- Jane Street Core for OCaml (industrial)
- CakeML Basis library (verified)

Modules vs. type classes

- type class inference selects instances from global database
- usually, only one instance per parameter allowed
  - sorting can't have both quicksort and mergesort instances
- signatures can have multiple implementing modules
- modules usually provide no automation for instantiation