DD2552 Seminar 8: Abstract types

Karl Palmskog

KTH

Thursday September 21, 2023

Course material

- PFPL Chapter 17
- CakeML example of abstracted queue implemented with two lists
- Bonus reading, "On Understanding Data Abstraction, Revisited", https://www.cs.utexas.edu/~wcook/Drafts/2009/essay.pdf

Data abstraction

- interfaces are a form of agreement between implementor and client of a program
- interfaces should isolate the client from the implementor
- 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 unspecified type
- two mechanisms:
  - information hiding (for client)
  - compliance checking (for implementor)
- example from last seminar: unspecified finite map datatype with operations `add`, `remove`, `find`, `mem`

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 $\rho \, \mathrm{with} \, e \, \mathrm{as} \, \exists(t.\tau)$
  - ρ is a type ("representation type")
  - e is expression of type $[\rho/t]\tau$ ("implementation")
- to use a package, use elimination form open $e_1 \, \mathrm{as} \, t \, \mathrm{with} \, x : \tau \, \mathrm{in} \, e_2$
  - 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, imperative

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

val create : unit -> 'a t
(* Return a new stack, initially empty. *)

val push : 'a -> 'a t -> unit
(* [push x s] adds the element [x] at the
 top of stack [s]. *)

val pop : 'a t -> 'a
(* [pop s] removes and returns the topmost element
   in stack [s], or throws an exception. *)
```

Existential types in OCaml, pure

```
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 "unaffected"
- informally, implementations are bisimilar when observers (clients) can't tell them apart by interacting with them

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