Invariants

October 30th

Outline

- Using invariants
- (if time permits: about the performance of BatDeque)
- Using parsers in ocaml

Invariants: motivation

- Sometimes we want to pre-compute a value:
  - The 'length' function on lists is expensive…
    - but keeping a running total is cheap
    - (the same applies for 'sum' and other recursively defined values)
  - The 'depth' function on trees
    - sort of expensive, but used a lot in balanced trees
- Oftentimes the entire structure needs to satisfy a property:
  - Sortedness of trees to speed up search
  - Normalization of numeric values (divisor of a fraction, first coefficient of a polynomial, …)

Running example

- Let’s take another look at double ended queues.
- These are fun because:
  - They are a real implementation
  - The same deque has multiple representations
  - They are easy to explain
  - There’s an invariant, too!

Actual code for deques (sorry for using record syntax!)

```
type 'a dq = { front : 'a list ; flen : int ;
               rear : 'a list  ; rlen : int }

let invariants t =
  assert (List.length t.front = t.flen);
  assert (List.length t.rear = t.rlen)

let empty = { front = [ ] ; flen = 0 ;
  rear  = [ ] ; rlen = 0 }

let size q = q.flen + q.rlen

let cons x q =
{ q with front = x :: q.front ; flen = q.flen + 1 }
```

Code for deques without record syntax

```
(* List and its length: *)
type 'a llist = LList of ('a list * int)
(* Front and rear: *)
type 'a dq = ('a llist * 'a llist)

let front (LList (lst, _), _) = lst
let rear (_,LList (lst, _)) = lst
let flen (LList (_, n), _) = n
let rlen (_, LList (_, n)) = n

let invariants t =
  assert (List.length (front t) = flen t);
  assert (List.length (rear t) = rlen t)
```

Creating and manipulating dequeues

```
let empty = (LList ([], 0), LList ([], 0))

let size q = flen q + rlen q

let cons x q
 = (LList (x::(front q), flen q + 1), snd q)

(* What proof-conditions does this give us,
   given that this is the resource invariant: *)
let invariants t =
  assert (List.length (front t) = flen t);
  assert (List.length (rear t) = rlen t)
```

Creating and manipulating dequeues

```
let empty = (LList ([], 0), LList ([], 0))
```

- Function ‘empty’ creates a dequeue:
  - Prove that it satisfies the invariant.
  - Proof conditions:
    - List.length (front empty) = flen empty
    - List.length (rear empty) = rlen empty
- These are very simple proofs (unfolding definitions)

Creating and manipulating dequeues

```
let size q = flen q + rlen q
```

- Function ‘size’ takes a dequeue, returns an int:
  - Proving its correctness might involve *using* the invariant …
  - … but there is nothing to prove to *establish* the invariant

Creating and manipulating dequeues

```
let cons x q
 = (LList (x::(front q), flen q + 1), snd q)
```

- Function
  cons : ‘a -> ‘a dq -> ‘a dq
- … was listed as a ‘constructor’, so we might be tempted to try and prove:
  - List.length (front (cons x q)) = flen (cons x q)
- this property does not hold without assumptions!

Creating and manipulating dequeues

```
let cons x q
 = (LList (x::(front q), flen q + 1), snd q)
```

- Function
  cons : ‘a -> ‘a dq -> ‘a dq
- … requires two proofs (one for each invariant). This:
  - List.length (front (cons x q)) = flen (cons x q)
  - under the assumptions:
    - List.length (front q) = flen q
    - List.length (rear q) = rlen q

Creating and manipulating dequeues

```
let cons x q
 = (LList (x::(front q), flen q + 1), snd q)
```

- Function
  cons : ‘a -> ‘a dq -> ‘a dq
- … requires two proofs (one for each invariant). And this:
  - List.length (rear (cons x q)) = rlen (cons x q)
  - under the assumptions:
    - List.length (front q) = flen q
    - List.length (rear q) = rlen q
  - (this proof is nearly immediate)

Why assume all invariants?

- For ‘cons’, we didn’t really need the assumption about the rear when doing the proof about the front, and vice versa.
- This frequently happens, but there are exceptions:

```
let rev q = (snd q, fst q)
```

  - Here which assumption is needed for which proof switches.

- So what’s the general pattern?

Resource invariants…

- Suppose the resource invariant is described by:
  invr : t -> bool
- Then for a function f:
- Assume that the invariant holds for all arguments to f of type t
- Prove that the invariant holds for all results of f
- Example, suppose f : x -> t -> t, then:
  - assume: invr b
  - prove: invr (f a b)

Q: What if there's a resource invariant…
… AND a canonical form function?

- invr : t -> bool
- cf : t -> t

- Do we get to assume invr for the argument of cf?
  - …
- Do we need to prove that
  '(cf x = cf y) implies (invr x = invr y)'?

Q: What if there's a resource invariant… … AND a canonical form function?

- invr : t -> bool
- cf : t -> t

- Do we get to assume invr for the argument of cf?
  - Yes! (more assumptions makes easier proofs)
- Do we need to prove that
  '(cf x = cf y) implies (invr x = invr y)'?

Q: What if there's a resource invariant… … AND a canonical form function?

- invr : t -> bool
- cf : t -> t

- Do we get to assume invr for the argument of cf?
  - Yes! (more assumptions makes easier proofs)
- Do we need to prove that
  '(cf x = cf y) implies (invr x = invr y)'?
  - No! All values that can be created satisfy invr
    - … hence 'invr x = true = invr y'

Example on deques

- Valid cf:

```
let cf x =
  let lst = front x @ List.rev (rear x) in
  (LList (lst, List.length lst), LList ([],0))

let eq x y = (cf x = cf y)

Hence:

eq x y = ((front x @ List.rev (rear x)) =
          (front y @ List.rev (rear y)) )
```

Example on deques

- Valid cf:

```
let cf x =
  let lst = front x @ List.rev (rear x) in
  (LList (lst, List.length lst), LList ([],0))
```

  - Q: Should we use ‘invr’ to make this more efficient?

Example on deques

- Valid cf:

```
let cf x =
  let lst = front x @ List.rev (rear x) in
  (LList (lst, List.length lst), LList ([],0))
```

  - Q: Should we use ‘invr’ to make this more efficient?
  - A1: This function will never get run
  - A2: This function is for use in proofs, might be easier if we don’t rely on ‘invr’

Example 2 on queues

- Valid abstraction function af:

- let af x = front x @ List.rev (rear x)

- Gives rise to the same ‘eq’ relation.
- Again, we don’t rely on invariants to implement this.

Complete aside

- … I want to address something about batteries' deque implementation

- … It has nothing to do with the rest of this lecture

A problem with deque in ocaml

- A functional data-structure can be ‘copied’ in O(1)
- The deque claims its operations are:
  - O(1) amortized (average)
  - O(n) worst-case

- Using functional data-structures, this is never really possible!

A problem with deque in ocaml

```
#require “batteries”
open Batteries

let rec downfrom i
 = if i = 0 then [] else i::downfrom (i-1)
let dq = BatDeque.of_list (downfrom 100000)
let dqr
 = List.map (fun _ -> BatDeque.rear dq)
            (downfrom 100000)
```

This takes a long time!!
Culprit is that ‘rear’ takes O(n) *every time*!

Fixing batteries…

- Memoizing can fix the issue (at the expense of memory overhead)
- A more complicated data-structure can fix the issue
- Using deques as linear types fixes the issue
    - This means that after a deque is passed as an argument, you don't get to use it elsewhere. I.e.: don't use the O(1) copy feature

A problem with deque in ocaml

```
#require “batteries”
open Batteries

let rec downfrom i
 = if i = 0 then [] else i::downfrom (i-1)
let dq = BatDeque.of_list (downfrom 100000)
let dqr
 = List.map (let r = BatDeque.rear dq in
              (fun _ -> r))
            (downfrom 100000)
```

This is ready in a blink of an eye

A problem with deque in ocaml

```
#require “batteries”
open Batteries

let rec downfrom i
 = if i = 0 then [] else i::downfrom (i-1)
let dq = BatDeque.of_list (downfrom 100000)
let dqr
 = List.map ((fun _ ->
              let r = BatDeque.rear dq in r))
            (downfrom 100000)
```

This takes ‘forever’

A problem with deque in ocaml

```
#require “batteries”
open Batteries

let rec downfrom i
 = if i = 0 then [] else i::downfrom (i-1)
let dq = BatDeque.of_list (downfrom 100000)
let dqr
 = List.map (fun _ -> BatDeque.rear dq)
            (downfrom 100000)
```

Old code for reference

Why doesn't ocaml do this automatically?

- Where you put your let is:
  - an easy change to make
  - a tradeoff between memory use and time

- Re-computing simple values is often fast
- Saving memory typically saves timely cache-misses
- Ocaml allows the programmer to make the tradeoff
  - … with a minimal change to the program itself

Using a parser

- In your final project, you'll build a parser.
- We'll use 'menhir'
- Ocaml uses the traditional lexer+parser generator
- What is a lexer:
  - Takes a string (or file)
  - Delivers a list of tokens
    (e.g. symbols, variable-names and keywords)
- What is a parser:
  - Takes a list from the lexer
  - Delivers a tree structure called parse-tree

On a parser

- Takes a grammar
- Checks that the input satisfies the grammar
- Produces a parse tree

- … for tomorrow's lab, you're given a very simple lexer and grammar:
  - just produce a list of words
  - we'll set everything up so that you have a working starting project

Installing and using menhir

- … we can do this with ‘opam’…
- There’s something robust that is more suitable for your final project: dune
  - Your ‘dune-project’ file will read something like:
    ```
    (lang dune 2.9)
    (using menhir 2.1)
    ```
- The directory with your parser gets a file called ‘dune’ with: (menhir (modules parser)) (ocamllex lexer)
- Compile your project: dune build
- Run your project by finding the executable
- Start utop within your project: dune utop srcdir

Tomorrow’s lab

- Following the instructions should get you through it rather quickly
- If you have ocaml working on your private computer, try and set up dune and menhir there as well
- Start forming groups for the final project:
  - Minimum of 2 people
  - Maximum of 3 people
- Use the remaining lab time to study for your midterm

Course outline

- Wednesday: ask me anything (optional lecture)
- Friday: second midterm
- As of next week: project
  - Parsing, project groups become official
  - Rewriting
  - Testing
  - Midterm
  - Bit of time to work more on project