Modular Reasoning

COS 326: Functional Programming

November 7, 2012

1 What can type systems do?

1.1 Express invariance about values

v: bool	then v = true or v = false
v: char	then v = 'a' or v = 'b' or...
v: list	then v = [] or v = hd::tail and hd: T and tail: T list
v: T_1 * T_2	then $v = (v_1, v_2)$ and $v_1 : T_1$ and $v_2 : T_2$
v: T_1 -> T_2	then v is a function and if you assume its input v_1 satisfies the invariants of T_1 and v v_1 -> v_2 then v_2:T_2

1.2 Enable abstraction

Actually a series of bits, not "true". But actually actually wires and signals and such. Quarks and "what's that boson thing they just discovered."

1.2.1 Relationship

Abstraction is a relationship between two worlds, imaginary and concrete.

2 Boolean module

```
module B :BOOL = struct
    type b = int
    let tru = 1
    let fal = 0
    let not b =
        match b with
        | 0 -> 1
        | 1-> 0
        | _ -> raise BrokenRepInv
```

```
        // satisfies because guaranteed only 0 or 1 will come in
    ;;
    let and bs =
        match bs with
        | (0, 0) | (0, 1) | (1, 0) -> 0
        | (1, 1) -> 1
        | (_, _) -> raise BrokenRepInv
end
```

2.1 Invariant

v: B.b then v = 1 or v = 1
defining a type b that will always be 1 or 0. claiming it's true; must check that everything satisfies it

2.2 Proof

tru	according to signature has type b; has to be either 1 or 0; is 1, so ok.
fal	as above
not	: b ->b. pick input v_1. Assume v_1:b. Show v v_1 -> v_2 and v_2 satisfies the invariants of b.
and	b*b ->b. Assume arg v. Assume v: b * b. Prove: and v ->v' and v':b.

2.3 Moral of the story

To check that your module satisfies a representation invariant, for all operations assume the rep inv holds for all inpurs. Prove it holds for all outputs.

3 Sets

3.1 Representation 1: Duplicates

list. represents particular set if members of the list are the same as members of the set.

3.2 Representation 2: No Duplicates

Lists, but only those without duplicates. e.g. [1,1] is not a set.

3.3 Implementation 1: Duplicates

```
module Set1 : SET = struct
  type 'a set = 'a list
  let empty = []
```

```
  let add x l = x::l
  let size l =
    match l with
    | [] -> 0
    | hd:: tl ->size tl + (if List.mem hd tl then 0 else 1)
  ...
end
```

3.4 Implementation 2: No Duplicates

```
module Set1 : SET = struct
  type 'a set = 'a list
  let empty = []
  let add x l =
    if List.mem x l then l
    else x::l
  let size l = List.length l \\ exploiting representation invariant
  ...
end
```

3.5 Proving stuff

The stronger the representation invariant, the more stuff you have to prove.

4 Protect from Client

module SET	client
type 'a set	set, set, set...
v: 'a set	sets are abstract
	no way to inject bad code

5 Back to Bool

```
module S: BOOL = struct
  type b = bool
  let tru = true
  let fal = false
  let not b =
    match b with
    | true -> false
    | false -> true
```

```
  let and bs =
    match bs with
    | true, true -> true
    | _, _ -> false
end
```

5.1 Mapping

Some concrete things represent imaginary ones. `not` maps an imaginary object to another imaginary object. We must make sure out implementation maps a related input to a related output.

5.2 Proof on our abstract types

Show that the abstraction function is correctly implemented. C $\rightsquigarrow$ a:b f $\rightsquigarrow$ f : t1 $\rightarrow$t2
Assume a pair of inputs c, a such that c $\rightsquigarrow$a:t1.
Must prove f c $\rightsquigarrow$ g a :t2

5.3 What?

To prove a module M1 faithfully implements a spec S, show that every element of the module is related like that (above).

5.4 Let's do it?

5.4.1 Step 1

1 $\rightsquigarrow$ true :b
0 $\rightsquigarrow$ false:b
tru $\rightsquigarrow$ tru:b
iff 1 $\rightsquigarrow$ tru : b
iff 1 $\rightsquigarrow$ true : b
iff valid

5.4.2 Step 2

Show: f $\rightsquigarrow$ fal: b
iff 0 $\rightsquigarrow$ false :b
iff valid

5.4.3 Step 3

Show: not $\rightsquigarrow$ not : b $\rightarrow$ b
Asume on inputs such that
c $\rightsquigarrow$ a :b
Must prove not c $\rightsquigarrow$ not a : b

case a = true
Assumption looks like:
c $\rightsquigarrow$ true :b
By definition of $\rightsquigarrow$
Therefore c = 1
Must prove n 1 $\rightsquigarrow$ not true: b
iff 0 $\rightsquigarrow$ not true:b
iff 0 $\rightsquigarrow$ false :b
iff valid!

case a = false
Assumption looks like:
c $\rightsquigarrow$ false
therefore c = 0
must prove:
not 0 $\rightsquigarrow$ not false
1 $\rightsquigarrow$ true
valid

5.4.4 Step 4

and $\rightsquigarrow$ and : b * b $\rightarrow$ b
Assume we have an input
c $\rightsquigarrow$ a : b * b
That means
c = (c1, c2)
a (a1, a2)
and
c1 $\rightsquigarrow$ a1 : b
and
c2 $\rightsquigarrow$ a2 : b
Must prove:
and (c1, c2) $\rightsquigarrow$ and (a1, a2) : b
Cases $\rightarrow$ and applied to any combination gives a result related to the result that and

produces.

6 Final morals

Reasoning about representation invariants and abstraction relations based on types.

6.1

c : Abs then we show RI(v) (module writer gets to pick) (representation invariant of v holds)

6.2

c $\rightsquigarrow$ a : Abs (module writer gets to pick the abstraction function)

6.3

f : Assume RI(inputs), Show RI(outputs)

6.4

f: Assume inputs are related, Show outputs are related

6.5 Logical Relations

From relation to implication. Assume input, show output.

6.6 Module Comments

In module comments, say what the abstraction relation is and what the representation invariant is.