CLU Review

CLU uses call by sharing, which is the same as called by pointer value (except that every identifer in CLU is implicitly bound to a pointer).

Notions of equality:

- `equal`: two objects are equal if they currently have the same value and will always have the same value (even if mutated). This is implemented as pointer equal on mutable objects and structural equality and immutable ones.
- `similar`: two objects are similar if they currently have the same value. This is typically implemented as structural equality.
- `similar1`: two container objects are similar1 if the currently have the same value (structure) and the elements within are `equal`. This is typically implemented as structural equality on the container with pointer equality on elements.

CLU Review

Notions of copying:

- assignment – after assignment, the two objects are equal
- copy – after copying, the two objects are similar
- copy1 – after copy1'ing, the two objects are similar1

Note: structural equality may actually be something more complicated, such as sorting or other normalization operations.

The distinction between mutable and immutable objects is key: this is an abstract notion, so "benevolent side-effects" do not count.

Representation Invariants

Given a type T and some representation (implementation) R, a *representation invariant* is a set of properties that always hold for any object of type R than represents an object of type T.

The representation invariant, $\mathcal{I}$, is a predicate on objects of the representation type:

$$\mathcal{I} : \texttt{rep\_type} \rightarrow \texttt{bool}$$

For example, databases, operating systems, compilers, and windowing systems, etc., typically have invariants that must be maintained on their large complex data structures.
E.g., a set represented as an array may have invariants like: the array is sorted and contains no duplicate elements.

Representation Invariant

Example: A linked list with a "memoized" copy of the list as a sequence, for use in conversion `ll2seq`.

```
rep = record [lst: variant[empty: null, full: link]
              valid_copy: bool,
              seq_copy: sequence[t]]
link = record [value: t, rest: linked_list[t]]
```

Given a representation objects `r`, the representation invariant is:

$\mathcal{I}$(`r`) = if `r.valid_copy` then same_elts(`r.lst, r.seq_copy`)
where same_elts(l, s) = `size(lst) = size(s)` &
$\forall$ i, $0 < i <=$ size(s) `nth(l,i) = s[i]`

Notation: the functions used in the representation invariant are either in the program or should be defined. The above assumes that `nth` and `size` are defined on lists and `fetch` and `size` are defined on sequences.

Representation Invariant

Example: Binary Search Tree

```
rep = variant[empty: null, full: node]
node = record [value: t, left: bst, right bst]
```

$\mathcal{I}$(`r`) = if `is_full(r)` then
    let `lt = get_full(r).left, rt = get_full(r).right` in
        $\mathcal{I}$(lt) & $\mathcal{I}$(rt) &
        if (`is_full(lt)` & `is_full(rl)`)
        then ((`get_full(lt).value` $<$ `get_full(r).value`)
            (`get_full(rt).value` $>$ `get_full(r).value`))

If this were a balanced tree, then we would add this property to the $\mathcal{I}$. Note that invariants implied by the representation type itself are not included.

Abstraction Functions

Given a type T and some representation (implementation) R, an *abstraction function* maps objects of the concrete type R to objects of the abstract type T.

$$\mathcal{A} : \texttt{rep\_type} \rightarrow \texttt{abstract\_type}$$

E.g., if we have implemented sets using arrays, then the abstraction function tells us what set corresponds to each possible array. The abstraction function is a partial function: it is only defined on objects for which the representation invariant holds.

Note: We sometimes have a bit of a notational problem, since we don't have a way of writing down abstract objects very well.

Abstraction Functions

Example: A linked list with a "memoized" copy of the list as a sequence, for use in conversion `ll2seq`.

```
rep = record [lst: variant[empty: null, full: link]
              valid_copy: bool,
              seq_copy: sequence[t]]
link = record [value: t, rest: linked_list[t]]
```

$\mathcal{A}$: `rep` $\rightarrow$ `linked_list`
$\mathcal{A}$(`r`): if `is_empty(r.lst)` then *nil*
    else *cons*(`get_full(r.lst).value`,
        $\mathcal{A}$(`get_full(r.lst).rest`))

This assumes that *cons* and *nil* are well-defined symbols.

Important questions: Which fields in the representation affect the abstract value? How? Which concrete values map to the same abstract value?

Representation Exposure

Representation exposure allows the user/client of an abstraction to modify the internal (concrete state), thereby violating the representation invariants.

1. A linked list with a conversion to a mutable array. If we memoize the array and then return (a pointer to) it, this allows the user to violate the representation invariant.
2. A set represented as an sorted array, if a conversion to array function returns the array itself.
3. An (acyclic) linked list in which the user can access and modify the pointers between nodes.

Representation Exposure

- It is the job of the data type implementor, not the client, to ensure that the representation is not exposed.
- Is representation exposure always a bad idea?
- Since many operations (member functions) return values that are inside the abstraction, how do you avoid exposure?
  1. Return a copy of internal data.
  2. Return an immutable version of the data (i.e., use a type that omits the mutators).
  3. Return a type for which all possible values maintain the internal representation invariant.

Type Invariants

Given a type T, a `type invariant` is a property that holds of any abstract object of type T.

Type invariants are axioms on the abstract type, which are independent of the implementation, but would be proven by looking at the implementation. The representation invariants are used to help prove the type invariants.

We often think of these invariants as simple properties on a data type, but in general, these invariants form the specification of the type.

- For all arrays, a is: $size(a) \geq 0$.
- For all stacks, s: $pop(push(s)) = s$.

Formal Specifications

We will study formal specifications for two reasons:

- They are needed for giving rigorous correctness proofs of complicated algorithms or systems.
  1. architecture, e.g., scoreboarding and cache consistency mechanisms
  2. operating systems, e.g., distributed services, synchronization
  3. databases, e.g., transaction processing
  4. compilers, e.g., correctness of transformations
- Understanding how to write them also helps in writing information specifications.

For beginners, the hardest part is often avoiding specifications that map one object to another that we already understand.

A Linked List Specification

nil:	$\rightarrow$ List
cons:	Elem $\times$ List $\rightarrow$ List
isempty:	List $\rightarrow$ Bool
car:	List $\rightarrow$ Elem
cdr:	List $\rightarrow$ List
append:	List $\times$ List $\rightarrow$ List

The axioms will be a set of equations about the list values. There are two properties of the axioms we wish to ensure:

1. consistency: that we haven't over-constrained the values so that no legal objects (or only a single uninteresting one) is possible.
2. completeness: that we have given enough axioms.
3. adequacy: do we have enough observers to distinguish unequal values.

A Linked List Specification

There are three kinds of operations associated with each new type:

1. constructors: these are the set of functions that return the new type and are required to generate all possible values. Normally, we also choose a minimal such set.
2. extenders: functions that also return values of the new type, but are not "essential" in that every value can be expressed without them.
3. observers: these functions return a value of a different (usually simpler) type.

Axioms for a Linked List

Here are axioms for the list operations.

isempty(nil) = true
isempty(cons(e, l)) = false

car(nil) = error
car(cons(e, l)) = e

cdr(nil) = error
cdr(cons(e, l)) = l

append(nil, l) = l
append(cons(e,l1), l2) = cons (e, append(l1, l2))

Axioms for a Linked List (continued)

We could also add size and nth, as needed in the previous abstraction function.

nth(l,1) = car(l)
nth(l,n) = if (n = 1) then car(l)
  else if n< 1 then error
  else nth(cdr(l),n-1)

size(nil) = 0
size(cons(e,l)) = 1 + size(l)

Symbol Table

init:	→ Symtab
enterblock:	Symtab → Symtab
leaveblock:	Symtab → Symtab
add:	Symtab × Id × Attrs → Symtab
is_inblock:	Symtab × Id → Bool
retrieve:	Symtab × Id → Attrs

leaveblock(init) = error
leaveblock(enterblock(s)) = s
leaveblock(add(s, i, a)) = leaveblock(s)

is_inblock(init, i) = false
is_inblock(enterblock(s), i) = false
is_inblock(add(s, i1, a), i2) = if (i1 = i2) then true else
  else is_inblock(s,i2)

retrieve(init, is) = error
retrieve(enterblock(s), i) = retrieve(s, i)
retrieve(add(s,i1,a), i2) = if (i1 = i2) then a else retrieve(s, i2)