CLU and Data Abstraction

Motivation for CLU was better support for programming in the large. Influenced later languages, in particular, ADA.
Requirements for languages that support data abstraction:

1. Linguistic construct to implement data abstraction as a single unit. The construct must allow for definition of the representation and function implementations.
2. Language must limit access to the represention to just the operations (i.e., methods) on that data type.

In languages like C or Scheme, one can develop a style that makes use of data types as separate modules. Does this work just as well?

Types of Abstraction

The CLU paper identifies three forms of abstraction:

1. procedural abstraction
2. data abstraction
3. control abstraction

What other kind of control abstraction are there? How are they “simulated” in languages that don’t provide direct support?

Data Abstraction in CLU: Example

```
linked_list = cluster [t: type] is new, append, first
                   rest, ll2seq, equal, similar, copy

    rep = variant[empty: null,
                  full: link]
    link = record [value: t,
                   rest: linked_list[t]]

    new = proc () returns (cvt)
        return (rep$make_empty(nil))
        end new
    first = proc (ll: cvt) returns (t) signals (empty)
        tagcase ll
           tag empty: signal empty
           tag full(l: link):
               return(l.value)
           end
        end first
```


CLU: Example continued

```
    rest = proc (ll: cvt) returns (linked_list[t])
                          signals (empty)
        tagcase ll
           tag empty: signal empty
           tag full(l: link):
               return(l.rest)
           end
        end rest
```

Points to note about CLU type system:

- Support for polymorphism – the linked list is a linked list of anything.
- Recursive types – a linked list type is recursive. Need for variant type to make these work.
- Signals (exceptions) are part of the function interface.
- A type interface is the name, plus the function interfaces of all operations on the type.

CLU: Up and Down

In CLU, the conversion between a concrete and abstract type is done using the `up` and `down` functions, which produce no executable code.

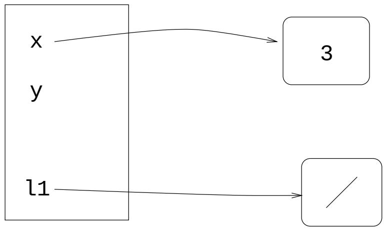
```
append = proc (ll: cvt, elem: t)
    tagcase ll
       tag empty:
           rep$change_full(ll, link${value: elem,
                                     rest:new()})
       tag full:
           append(rest(up(ll)), elem)

       end
    end append
```

The `cvt` type stands for “convert.” As a type of a formal paremeter, it means there is an implicit `down` operation; as a return type, it means there is an implicit `up` operation.

CLU Object Model

All variables in CLU are identifiers that may bound to objects. Object are created (implicitly, by calling any function that creates objects of the given type). They are destroyed (garbage collected) by the system.

```
x: int = 3
y: int
l1: linked_list[int] = linked_list$new()
```



Iterators

```
elements = iter (ll: cvt) yields (t)
    tagcase ll
       tag empty: return
       tag full(l: link):
           yield(l.value)
           for elem: t in elements(l.rest) do
               yield(elem)
               end
       end
    end elements
```


Safe Conversion Between Types

```
t_arr = array[t]
t_seq = sequence[t]

ll2seq = proc (ll: cvt) returns (t_seq)
    at: t_arr := t_arr$new()
    for elem: t in elements(up(ll)) do
        t_arr$addh(at, elem)
        end
    return(t_seq$a2s(at))
    end ll2seq
```

We could also do this by keeping an internal “memoized” result for the conversion function, along with a flag (either as a variant or separate boolean) to indicate whether the sequence is valid.

Equality

Two notions of equality: This the `eq?/equal?` distinction in Scheme.

1. *Structural equality:* Currently the same, but may be different in the future.
2. *Object (or pointer) equality*: The same object – if one is mutated, they will "both" be mutated in the same way.

```
equal  = proc (ll1, ll2: cvt) returns (bool)
    return(ll1 = ll2)
    end equal

similar = proc (ll1, ll2: cvt) returns (bool)
            where t in similar_type
    return(rep$similar(ll1, ll2))
    end similar
```

Basic observation: for immutable types, one cannot tell the difference between these two, so they are both defined as structural equality.

Copying

For copying, there are also two different notions:

1. The assignment statement in CLU does pointer copying.
2. By convention, the `copy` operation does a structural copy.

```
copy = proc (ll: cvt) returns (cvt)
            where t in copy_type
    return(rep$copy(ll))
    end copy

end linked_list   % this is the bottom of the scope
                  % opened with the cluster keyword
```

Are there cases in which a `copy` operation may simply do a pointer copy, and still have the same semantics?

Equal, Similar and Copy on Polymorphic Types

The above notions of equality, copying, and assignment are complete for objects, but other notions make sense for containers of other objects, i.e., polymorphic types in which the container is visible to the user.

```
similar1 = proc (ll1, ll2: cvt) returns (bool)
        % no where clause needed
    return(ll1 = ll2)
    end similar1

copy1 = proc (ll: cvt) returns (cvt)
        % no where clause needed
    return(rep$copy1(ll))
    end copy
```


Syntactic Sugar in CLU

Every user-defined operation in CLU has a complete name of the form:

```
type_name$function_name
```

Some functions names with appropriate type signatures also have a "sugared" form that can be used to invoke the function, e.g.,

```
l1 = l2       ->     linked_list$equal(l1, l2)

l1[3]         ->     linked_list$fetch(l1, 3)

l1 + l2       ->     linked_list$add(l1,l2)
```

Note: `l1 := l2` is assignment.

Unlike most object-oriented languages, there is no implicit argument to each operation, and no requirement that the first argument type be the class type. This simplifies the use of binary operators.

Iterators

```
elements = iter (ll: cvt) yields (t)
    tagcase ll
       tag empty: return
       tag full(l: link):
           yield(l.value)
           for elem: t in elements(l.rest) do
               yield(elem)
               end
       end
    end elements
```


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.

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

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.

Representation Exposure

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