COMP 304 2000 outline answers

Neil Leslie

October 20, 2003

1 Preamble

These are very brief notes on what I expected as answers. Many of the question in COMP 304 exams are 'book work'. In these cases I have simply stated this. It should be apparent from the question what part of the lecture notes the question refers to.

Questions

(1) Parts (a), (b) and (c) are basically book-work.

Part (d) consists of presenting a sensible variant of the rules presented in the lecture notes.

Part (e) is also basically book work.

The answer for part (f) consists of the rather obvious statement that programs are made form the stuff in programming languages, and the assertion that axiomatic semantics, with its notions of pre- and post-conditions sits well to our informal notion of (imperative) programs as functions from states to states. Further there is (usually) a clear relationship between the conditions in the code, and the conditions in the logic. An example is useful.

(2) (a) A higher-order function is a function which ...

Benefits include:

- capture common patterns
- hence promote re-use, correctness
- functions as first-class citizens, function types not a special case, hence simplicity

- added expressiveness,
- allow programmer to extend language with new ‘control structures’
- more abstraction
- support for currying and partial application
- ...

(b)

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (h:t) = f h : map f t

compose :: (a -> b) -> (c -> a) -> c -> b
compose f g x = f (g x)

all, some :: [Bool] -> Bool
all [] = True
all (h:t) = h && all t

some [] = False
some (h:t) = h || some t
```

(c) An inductively defined type is a type in which values may be defined in terms of values of the type itself, e.g.:

```
data List a = Nil
            | Cons a (List a)
```

is inductively defined because in the second clause we see that a list may be constructed from a sub-list.

Recursively defined functions on inductively defined data types typically have 1) one clause for each constructor, 2) a recursive call on each inductive sub-part. So for the type of lists above we would expect:

```
foo :: (List a) -> b
foo Nil = ??
foo (Cons v l) = ??? v (foo l)
```

we can capture this regularity with a recursion operator:

```
rec :: b -> (a -> (List a) -> b) -> (List a) -> b
rec d _ Nil = d
rec d e (Cons a l) = e a (rec d e l)
```

(3) (a) Declarative semantics is the explanation of

```
A :- B, C.
A : D.
```

etc. in terms of A holds if B holds and C holds, or A Holds if D holds.

Operational semantics is expressed in terms of unification, the generation of sub-goals and depth-first search and backtracking...

Problems arise because these two may not match properly.

(b)

```
append([], L, L).
append([H | T], L, [H | TL]) :-
   append(T, L, TL).
```

Explain how unification, depth-first search and backtracking works.

(c)

Meta-programs are programs which manipulate programs. Any example from the lectures would do as an answer.

(4) (a) book-work

(b) p-b-v output 3 6 2 values 1 2 3, p-b-vr output 3 6 2 values 3 6 2, p-b-r output 3 6 2 values 3 6 2.

(c) Essentially it is too hard to understand due to the possibility of name clashes and hard-to-predict behaviour. Examples help.

(5) An opportunity to discuss your favourite programming language.