CSL 665: Introduction (Logic and Functional) Programming
I semester 2003-04

Major Sat 29 Nov 2003 VI 430 9:00-??:?? Max Marks 45

Notational conventions.

- $\triangleq$ denotes a definition of a name by a term (usually a λ-term)
- $\equiv$ denotes "syntactic identity". It is very often used to replace a name by its definition
- $\equiv_\alpha$ denotes α equivalence.
- For any function symbol f or predicate symbol p, $\alpha(f)$ and $\alpha(p)$ denote their respective arities.

---

1. It is possible to simulate various imperative language behaviours *functionally* by using the memory state as a parameter of a function. Define higher order functions in `SML` for the following imperative constructs which have their usual meaning (as in *Pascal*).

   (a) `while` *BooleanExpression* `do` *Command*

   (b) `repeat` *Command* `until` *BooleanExpression*

   (c) `for i:=` m `to` n do *Command*

   $3 + 3 + 4 = 10$ marks

   Solution.

   We will assume that memory state (consisting of all the imperative variables) is a function from *integers* to values of some kind. Each integer represents a memory address and hence some simple variable. The function is partial in the sense that there are memory locations which do not contain any meaningful value.

   A typical imperative *Command* changes state to a new state. Hence a *Command* as state transformer is a higher order function. On the other hand each expression (especially *Boolean-Expression*) is also a function that takes the state as an argument, but rather than changing state (we assume for simplicity that there are no side-effects) returns a boolean value.

   With this preamble we are now ready to define the 3 looping commands as ML functions.

   (a) Let `b` denote such a boolean expression, `c` a command, and `s` the state. Then we have

   ```
   fun whiledo b c (s:int -> 'a) = if (b s)
                                   then (whiledo b c (c s))
                                   else s
   ```

   Here `(b s)` denotes the application of the boolean expression to the state to yield a truth value. If the expression is false , the state is returned unchanged. Otherwise the while loop is evaluated in the new state obtained by the application of the command `c` to the state `s`.

   As an illustration of the use of this while loop we compute the factorial function to imitate the following piece of *Pascal* code.

   ```
   p := 1;
   i := 1;
   while (i <= n) do
   begin
   ```

```
    p := p * i;
    i := i + 1
end
```

The ML version looks like this:

```
fun fac_whiledo n s =
      let fun condition s = (s 1) <= n;
          exception Undefined;
          fun statechange s 1 = (s 1) + 1     (* i *)
            | statechange s 2 = (s 2) * (s 1) (* p *)
            | statechange s 3 = n             (* n *)
            | statechange s _ = raise Undefined;
          val finalstate =  whiledo condition statechange s;
       in finalstate (2)
       end
```

(b) We use the fact that a repeat until loop executes at least once and then behaves like a while loop with a negated condition, to obtain the following function.

```
fun repeatuntil b c (s:int -> 'a) = let val news = (c s);
                                        val negb = fn s => not(b s)
                                     in whiledo negb c news
                                     end
```

(c) We could have followed the conventional wisdom of using the while loop to simulate the for-loop too. However, keeping in mind the fact that while loops represent potentially indefinite computations whereas for-loops represent primitive recursion (defined by induction) we choose a different approach. Here the counting variable is a purely functional variable even though it may be used in updating other imperative variables. However, languages like *Pascal* (which attempt to promote discipline in the use of counting variables) do not permit their updation within the body of the loop. The net result looks something like this:

```
(* Program to calculate f(n, f(n-1, ...f(2, f(1,e))...)) *)

fun upto_n f n e = if n <= 0 then e    (* e denotes the state *)
                   else let val newe = upto_n f (n-1) e
                         in f(n, newe)  (* f is a command that takes n too *)
                        end;
fun for_loop f m n e = upto_n f (n-m+1) e;

(* Compute factorial n using the for-loop *)

fun fac_for_loop n = for_loop mult 1 n 1;
```

where `mult` denotes the binary multiplication operation.

2. (a) Let b denote any base type and let the following grammar define the language of simple types for the simply-typed lambda calculus

$$\tau ::= b \mid (\tau \rightarrow \tau)$$

Let Γ be a set of type assignments for variables. Then prove that substitution preserves types int he simply typed lambda calculus, i.e. if $x : \alpha$, $M : \alpha$ and $L : \beta$ then $L\{M/x\} : \beta$.

(b) Find the most general type assignment for the combinator $\mathbf{S} \triangleq \lambda xyz[((x\ z)\ (y\ z))]$ in the second-order language of types defined by

$$\begin{array}{rcl} \tau & ::= & b \mid t \mid (\tau \to \tau) \\ \pi & ::= & \tau \mid \forall t[\pi] \end{array}$$

where b denotes any base type and t denotes a type variable.

$5 + 5 = 10$ marks

<u>*Solution*</u>.

(a) We prove it by induction on the structure of λ-terms in the simply typed λ-calculus. Assume $x : \alpha$, $M : \alpha$ and $L : \beta$ where α and β are simple types.

$$\begin{array}{lcl} (x : \alpha)\{M : \alpha/x : \alpha\} & \equiv & M : \alpha \\ (y : \beta)\{M : \alpha/x : \alpha\} & \equiv & y : \alpha \\ (\lambda x : \alpha\ [N : \gamma] : \alpha \to \gamma)\{M : \alpha/x : \alpha\} & \equiv & \lambda x : \alpha\ [N : \gamma]\ : \alpha \to \gamma \\ (\lambda y : \beta\ [N : \gamma])\{M : \alpha/x : \alpha\} & \equiv & \lambda y : \beta\ [N\{M : \alpha/x : \alpha\} : \gamma]\ : \beta \to \gamma \\ (P : \beta \to \gamma\ \ Q : \beta) : \gamma & \equiv & (P\{M : \alpha/x : \alpha\} : \beta \to \gamma\ \ Q\{M : \alpha/x : \alpha\} : \beta) : \gamma \end{array}$$

The last few clauses hold by induction hypothesis.

(b) The proof starts with some assumptions using type variables. In the following proof assume r, s, t, u, v and w are type variables (not necessarily all distinct).

$$\begin{array}{l}
x : t, y : u,\ z : v \vdash z : v|\{\} \\
\qquad\qquad\qquad\quad \vdash (y\ z) : w \\
\qquad\qquad\qquad\quad |\{u = v \to w\} \\
\qquad\qquad\qquad\quad \vdash (x\ z) : s \\
\qquad\qquad\qquad\quad |\{t = v \to s, u = v \to w\} \\
\qquad\qquad\qquad\quad \vdash ((x\ z)\ (y\ z)) : r \\
\qquad\qquad\qquad\quad |\{s = w \to r, t = v \to s, u = v \to w\} \\
\qquad\qquad \vdash \lambda z[((x\ z)\ (y\ z))] : v \to r \\
\qquad\qquad\qquad\quad |\{s = w \to r, t = v \to s, u = v \to w\} \\
\quad \vdash \lambda y \lambda z[((x\ z)\ (y\ z))] : (v \to w) \to (v \to r) \\
\qquad\qquad\qquad\quad |\{s = w \to r, t = v \to s, u = v \to w\} \\
\vdash \lambda x \lambda y \lambda z[((x\ z)\ (y\ z))] : (v \to (w \to r)) \to (v \to w) \to (v \to r) \\
|\{s = w \to r, t = v \to s, u = v \to w\}
\end{array}$$

3. In general a skolem normal form need not be unique (that is there may be two or more different skolem normal forms which are not even logically equivalent to each other). Let $A(x, y)$ be a binary atomic predicate and $Q(x)$ be a unary atomic predicate.

   (a) Find <u>two</u> distinct non-equivalent normal forms for the predicate $\forall x \forall y[P(x, y) \to \forall z[Q(z)]$.

   (b) Define an interpretation which illustrates that the two Skolem normal for are <u>not</u> logically equivalent.

$6 + 4 = 10$ marks

<u>*Solution*</u>.

(a) Consider the following logical equivalences:

$$\begin{array}{rclr} p \to \forall z[q] & \Leftrightarrow & \forall z[p \to q] & \text{where } z \notin FV(p) \\ \forall x[p] \to q & \Leftrightarrow & \exists x[p \to q] & \text{where } x \notin FV(q) \end{array}$$

Depending upon the order of application of these two equivalences we get the following distinct prenex normal forms for $\forall x \forall y[P(x, y)] \to \forall z[Q(z)]$, since the conditions to prevent free variable capture are all satisfied:

$$\begin{array}{rl} & \forall x \forall y[P(x, y)] \to \forall z[Q(z)] \\ \Leftrightarrow & \forall z \exists x \exists y[P(x, y) \to Q(z)] \\ \Leftrightarrow & \exists x \forall z \exists y[P(x, y) \to Q(z)] \\ \Leftrightarrow & \exists x \exists y \forall z[P(x, y) \to Q(z)] \end{array}$$

(b) On Skolemization we get the following three Skolem normal forms respectively:

i. $\forall z[\neg P(f(z), g(z)) \vee Q(z)]$ where f and g are new unary function symbols.

ii. $\forall z[\neg P(c, g(z)) \vee Q(z)]$ where c is a new constant symbol and g is a new unary function symbol.

iii. $\forall z[\neg P(c, d) \vee Q(z)]$ where c and d are new constant symbols.

(c) It is easy to see that under the following interpretation the three Skolem normal forms are different. Let

- the domain of discourse be the set $D = \{1, 2, 3\}$,
- f be the constant function $f(\_) = 1$,
- g be defined as the function $g(1) = 2$, $g(2) = 3$ and $g(3) = 1$,
- c denote the element 3,
- d denote the element 2
- P denote the binary relation $\{(3, 1), (3, 2), (3, 3)\}$,
- Q denote the unary relation $\{1, 2\}$.

It is clear that

- (1) is true because $P(f(z), g(z))$ is always false
- (2) is false because for $z = 3$, $P(3, 1)$ is true whereas $Q(3)$ is false, but is true for $z = 1$.
- (3) is false because even though it is true for $z = 3$ it is false for $z = 1$

4. Let `derivative(Y, X, Z)` denote that the derivative of `Y` with respect to `X` is `Z`.Given the following facts:

```
derivative(N, X, 0).
derivative(X, X, 1).
derivative(sin(X), X, cos(X)).
derivative(cos(X), X, -sin(X)).
derivative(exp(X), X, exp(X)).
derivative(log(X), X, 1/X).
```

write logic programming rules to compute derivatives of

(a) sum of two expressions

(b) difference of two expressions

(c) product of two expressions

(d) quotient of two expressions

In particular make sure your definitions can calculate the derivatives of expressions such as $3 * exp(5 * X) * sin(2 * X) + cos(X)/log(X)$ 5 marks

<u>*Solution*</u>.

The *chain rule* for unary functions:

$$\frac{d}{dx} f(g(x)) = \frac{d}{dg(x)} f(g(x) \times \frac{d}{dx} g(x)$$

yields the following program. Note that we use the order of evaluation crucially. Hence the clause for constants and other expressions not dependent on the variable `X` has been moved to the last.

```
derivative(X, X, 1).
derivative(sin(X), X, cos(X)).
derivative(cos(X), X, -sin(X)).
derivative(exp(X), X, exp(X)).
derivative(log(X), X, 1/X).
derivative(sin(Y), X, cos(Y)*DY) :- derivative(Y, X, DY).
derivative(cos(Y), X, -sin(X)*DY) :- derivative(Y, X, DY).
derivative(exp(Y), X, exp(X)*DY) :- derivative(Y, X, DY).
derivative(log(Y), X, 1/X * DY) :- derivative(Y, X, DY).
derivative(plus(E, F), X, plus(DE,DF)) :- derivative(E, X, DE),
                                           derivative(F, X, DF).
derivative(minus(E, F), X, minus(DE,DF)) :- derivative(E, X, DE),
                                             derivative(F, X, DF).
derivative(mult(E, F), X, plus(mult(F, DE), mult(E, DF))) :-
                   derivative(E, X, DE), derivative(F, X, DF).
derivative(quot(E, F), X, quot(minus(mult(F,DE),mult(E,DF)),mult(F, F))) :-
    derivative(E, X, DE), derivative(F, X, DF).
derivative(N, X, 0).
```

5. Translate the following argument into First order logic and prove it using resolution.

Whoever visited the building was observed. Anyone who had observed Ajay, would have remembered him. Nobody remembered Ajay. Therefore Ajay did not visit the building.

10 marks

<u>*Solution*</u>.

Assume the following predicate symbols with their arities marked by the number of variable parameters.

- $V(x)$: "x visited the building",
- $O(z,w)$: "z observed w". Equivalently, "w was observed by z". The passive voice version "w was observed" is merely a short form for "w was observed *by someone*", which would be translated into First-order logic as $\exists z[O(z,w)]$.
- $R(u,v)$: "u remembered v". Even though we are treating remembered as a binary predicate[1], for the purpose of this particular argument it could have been treated as a unary predicate $R(u)$ with the meaning "u remembered Ajay" and it would have made no difference to the rest of the solution. By the same argument we could have treated "visited" as a binary predicate.

The argument now reads in First-order logic as follows[2]. We use the symbol a to denote the constant Ajay.

(a) $\forall x[V(x) \rightarrow \exists y[O(y,x)]]$.

(b) $\forall z[O(z,a) \rightarrow R(z,a)]$.

(c) $\forall u[\neg R(u,a)]$. Equivalently we could have translated it as $\neg\exists u[R(u,a)]$.

(d) Therefore $\neg V(a)$.

Combining all the hypotheses and negating the conclusion, we get the predicate

$$\forall x[V(x) \rightarrow \exists y[O(y,x)]] \wedge \forall z[O(z,a) \rightarrow R(z,a)] \wedge \forall u[\neg R(u,a)] \wedge V(a)$$

[1]Obviously, since "remembered" is a transitive verb and hence requires both subject and object in any natural language, except in the passive voice, where the object is implicit as in the case of "observed".

[2]Note that bound variables have been carefully kept unique for every hypothesis, this is important both for the purpose of Skolemization and since for resolution, all quantifiers are removed and one would not like implicitly bound variables to be confused

.

The quaontifier $\exists y$ is first moved left[3] to yield the logically equivalent form

$$\forall x \exists y[\neg V(x) \vee O(y,x)] \wedge \forall z[\neg O(z,a) \vee R(z,a)] \wedge \forall u[\neg R(u,a)] \wedge V(a)$$

which on further movement of quantifiers yields

$$\forall x \exists y \forall z \forall u[(\neg V(x) \vee O(y,x)) \wedge (\neg O(z,a) \vee R(z,a)) \wedge \neg R(u,a) \wedge V(a)]$$

.

Skolemizing with a unary function symbol f, we get the following normal form

$$\forall x \forall z \forall u[(\neg V(x) \vee O(f(x),x)) \wedge (\neg O(z,a) \vee R(z,a)) \wedge \neg R(u,a) \wedge V(a)]$$

.

Since it is in Skolem CNF now, we proceed with resolution on the body of the formula.

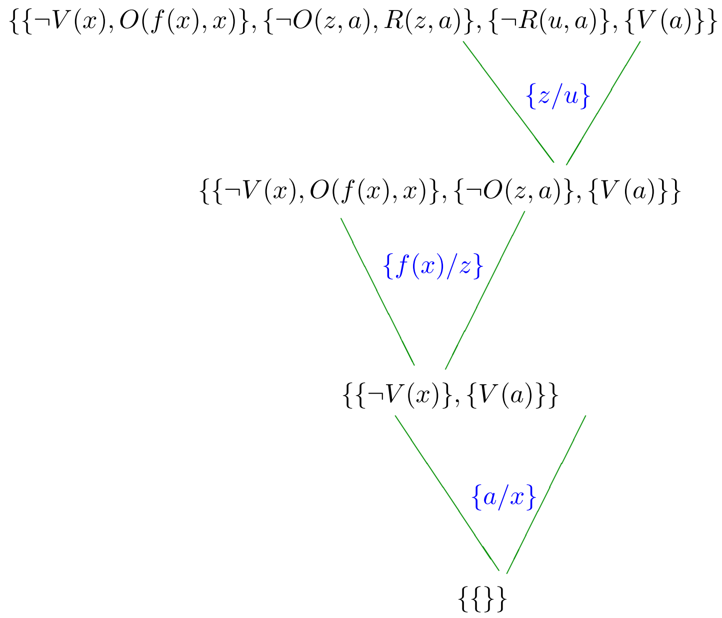

Figure 1: The resolution procedure

[3]we could have moved it later too, but the skolem normal form would be different!