Programming Languages and Compilers

Qualifying Exam
September 1995

You must use a different booklet for each of the 6 questions

1. (a) Define the term *parametric polymorphism* and give an example of a polymorphic function in ML, along with its type.

   Polymorphism refers to the ability for different instances of the same object, generally a function, to take on different types. Parametric polymorphism is the form of polymorphism in which the type of the object contains universally quantified type variables. These type variables can be instantiated by any type, and can be instantiated differently in separate uses of the object. For example, the length function

   ```
   fun length [] = 0
    |  length (x::xs) = 1 + length xs
   ```

   is polymorphic and has the type `'a list -> int`.

   (b) In what ways are parametric polymorphism in ML and dynamic overloading (i.e. the run-time determination of which virtual member function is being called) in C++ fundamentally different? Describe how these features are implemented in the two languages (if you are not sure exactly how it is done, any plausible implementation is fine). In particular, make sure you mention data representation and parameter passing.

   In ML, there is (almost) no overloading - all instances of the same identifier in the same scope refer to the same object. For example, the + in the length function cannot depend of the type of the arguments to length. As a consequence, any component of a formal parameter that corresponds to a type variable in the type of the formal parameter cannot be accessed. Thus, that component can be represented in any way desired, except that generally these components must all be the same size. For example, in ML, the `'a list` in the length function, for all types `'a`, is implemented as a list whose `car` fields are one word - either a one word value such as an integer or boolean, etc., or a pointer to a larger type.

   Dynamic overloading in C++, however, allows for identifiers within a function definition to refer to different values depending on the type of the function's actual parameters. In C++, these identifiers that are overloaded are the virtual member functions defined in the formal parameter's class (type). Since the overloading (being dynamic) depends on the type of the parameters, the definition of overloaded identifier must be passed at procedure call time. C++ implementations support this by including a pointer to each virtual member function for a class within the record representing values of the class (and its derived classes). Calls to the virtual member functions are implemented as an indirect call using those pointers.

   (c) In ML, equality is not defined for functions. That is, for two functions `f` and `g` the compiler will reject the expression `(f = g)`. What then, is the type of

   ```
   fun h x y = if x = y then 1 else 0
   ```

   (hint: `h` is not monomorphic, yet the expression `(h (fn a => a) (fn b => b))` is type-incorrect). If you are not sure exactly what the type would be in ML, then suggest how you would handle it if you were designing ML.

The equality operator (strangely) is the only dynamically overloaded operator in ML - and is thus an exception to part b) above. Clearly, different equality operators must be called depending on whether x and y are integers, reals, lists, etc. However, the only types that x and y can take on are those for which equality is defined, the so-called *equality types* (which is practically all types not containing functions). The type of f above is

```
''a -> ''a -> int
```

which is read as "for all equality types a, a to a to int".

(d) Describe how ML's polymorphic equality, as seen above, might be implemented.

As with virtual member functions in C++, either run-time type information or a pointer to the appropriate equality operator, must be passed when f is called. In SML/NJ, the values are tagged with type information which is used to call the appropriate equality operator.

2. (new booklet, please)

(a) State the Church-Rosser Theorem I (hint: it says something about two inter-convertible expressions e_1 and e_2)

If $e_1 \overset{*}{\Leftrightarrow} e_2$ then there exists an e_3 such that $e_1 \overset{*}{\Rightarrow} e_3$ and $e_2 \overset{*}{\Rightarrow} e_3$

(b) Prove the following corollary: For any expression in the lambda calculus there exists at most one normal form.

Suppose otherwise, that there is an expression e such that for two distinct normal forms e_1 and e_2, $e \overset{*}{\Rightarrow} e_1$ and $e \overset{*}{\Rightarrow} e_2$. By reversing the reduction, we see that $e_1 \overset{*}{\Leftrightarrow} e$ and $e_2 \overset{*}{\Leftrightarrow} e$, thus $e_1 \overset{*}{\Leftrightarrow} e_2$. By the Church-Rosser Theorem I, there must exist an e_3 such that $e_1 \overset{*}{\Rightarrow} e_3$ and $e_2 \overset{*}{\Rightarrow} e_3$. Thus, e_1 and e_2 cannot be distinct normal forms.

(c) Give an example of an expression with no normal form.

$$(\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))$$

(d) i. Is call-by-name in Algol more similar to normal order or applicative order reduction in the lambda calculus?

Normal order, since the Algol copy rule, like normal order reduction, specifies that the expression representing the actual parameter is first substituted for the formal parameter in the function body. Only when the formal parameter is needed in the function body will the actual parameter be evaluated.

ii. For any two terminating Algol programs that are identical except that one uses call-by-name and the other call-by-value, will they always produce the same result? If so, state why the designers of Algol chose to support both parameter passing mechanisms. If not, state why this is not a violation of the above corollary and give a sample program that illustrates your point.

No. Algol, being an imperative language containing assignment, is not based on the lambda-calculus. The Church-Rosser Theorem I and its corollary do not apply to it.

```
var y:integer;

procedure f(x: integer)
begin
```

```
  x := x + 1;
end

begin
  y := 3;
  f(y);
  print(y);
end.
```

3. (New booklet, please)
Control backtracking is backtracking which does not undo assignments or changes to data-structures. It can be defined in terms of two no-argument operations, defined as follows:

- `ok`: establishes a backtracking point; when evaluated (the first time) `ok` returns true, but if backtracked into returns false
- `fail`: backtracks to the most recent backtracking point, which it removes, and forces the corresponding `ok` to return false

Note that

```
if ok() then
        a(); fail();
else
        b();
end if
```

is equivalent to

```
a(); b();
```

if `a` and `b` contain no `ok` or `fail` calls.

(a) Show how control backtracking can be used to implement a backtracking parser for an ambiguous (non-left-recursive) context-free grammar. Hint: show how to write functions corresponding to each of the following three rules:

```
<t>     ::=     terminal
<s>     ::=     <c1>  <c2>
<a>     ::=     <c1>  |  <c2>
```

You may assume the input is a Lisp-like list of tokens, and you do not have to build a parse tree (i.e. write only a 'recognizer).

```
function t (l:list) =>
        if car(l) = "terminal" then cdr(l)
        else fail()
        end if;
function s (l:list) =>
        c2(c1(l));
function a (l:list) =>
        if ok() then c1(l)
        else c2(l)
        end if;
```

(b) Show how a parser for such a grammar can be written in Prolog.

```
t([terminal|R],R).
s(L,R) :- c1(L,R1), c2(R1,R).
a(L,R) :- c1(L,R).
a(L,R) :- c2(L,R).
```

4. (new booklet, please)
A naive class intlist for a linked list of integers is written in C++ as follows:

```
struct intnode {int val; void *next;};
class intlist
{       intnode *pin;
public:
        intlist(int i)
        {       pin = new intnode;
                pin->head = i;
                pin->tail = this;
        };
        int head() {pin->val};
        intlist tail() {pin->next};
        int null() {pin=0};
};
```

NOTE: The naive version was incorrect. It should have been something like:

```
struct intnode {int val; int *next;};
class intlist
{       intnode *pin;
public:
        intlist() {pin = 0}
        intlist(int i, intlist r)
        {       pin->val = i;
                pin->next = r;
        };
        int head() {return pin->val};
        intlist tail() {return pin->next};
        bool empty() {return pin==0};
};
```

(a) Show how templates can be used to generalize this definition to lists of arbitrary types.

```
template <class eltype>
class list<eltype>
{       struct node {eltype val; eltype *next;};
        node<eltype> *pin;
public:
        list<eltype>() {pin = 0}
        list<eltype>(int i, list<eltype> r)
        {       pin->val = i;
                pin->next = r;
        }
        eltype head() {return pin->val}
        list<eltype> tail() {return pin->next}
        int empty() {return pin==0}
```

```
};
```

(b) Show how this definition (not the solution to (a)) can be extended to provide automatic garbage collection of inaccessible nodes. Hint: use reference counts.

Need to specify copy and assignment operations:

```
class intlist
{       struct intnode {int val; int *next; int rc;};
        intnode *pin;
public:
        intlist() {pin = 0}             //empty list
        intlist(int i, intlist r)       //cons
        {       pin->val = i;
                pin->next = r;
                pin->rc = 1;
        }
        ~intlist()                      //destructor
        {       if (pin != 0)
                        if (pin->rc == 0) delete pin;
        }
        int head() {return pin->val};
        intlist tail()
        {       if (pin!=0) (pin->next->rc)++;
                retun pin->next;
        }
        bool empty() {return pin==0};
        intlist(const intlist &l)       //copy
        {       if (l.pin->rc != 0) (l.pin->rc)++;
                pin = l.pin;
        }
        op=(intlist &l)                 //assignment
        {       if (pin!=0)
                {       (pin->rc)--;
                        if (pin->rc == 0) delete pin
                }
                if (l.pin->rc != 0) (l.pin->rc)++;
                pin = l.pin;
        }
};
```

(c) Show how this definition is translated into C.

No virtuals, so no dispatch tables. Member functions take explicit first arguments, and overloading is resolved.

5. (new booklet, please)
Consider a programming language that reads external input in string form. An input data item can be an integer, or a record written in the form {s_1 `=>` v_1, ..., s_k `=>` v_k }, where s_i is an attribute (or selector) and v_i is its corresponding value for $i = 1, \ldots, k$. The empty record is written {}. A value can either be a record or an integer, and there is no restriction on the depth of nesting or on the types of values. Internally, two records are equal if they contain the same SET of attribute/value pairs. The order of presentation of attribute/value pairs in the input is irrelevant. A record can be presented with duplicate attribute value pairs, but such duplication

is removed before conversion to internal form. For example, records {`a => 1, b => 2`} and {`a => 1, b => 2, a => 1`} are equal. The input is invalid when two top-level attributes s_i and s_j, $i \neq j$, within the same record are the same identifier but their corresponding values are not equal.

(a) Write an LL(1) grammar for an input data item.

```
    I = int | `{' Ilist `}'
Ilist = id `=>' I nlist | lambda
nlist = `,' Ilist | lambda
```

(b) Make your grammar s-attributed for producing syntax trees. Your syntax trees should have no brackets or commas. Nodes should be tagged with the type of data that they represent- either INTEGER or RECORD.

```
    I = int |                 I.nptr = mkleaf(INTEGER,int.entry)
        `{' Ilist `}'         I.nptr = mknode(RECORD,Ilist.nptr)
Ilist = id `=>' I nlist |     Ilist.nptr = cons(mkpair(id.entry,I.nptr),
                                                nlist.nptr)
        lambda                Ilist.nptr = nil
nlist = `,' Ilist |           nlist.nptr = Ilist.nptr
         lambda               nlist.nptr = nil
```

(c) Given a syntax tree for correct input, sketch a way to detect and prune all duplicate attribute/value pairs in a record, and to detect invalid input. Your method must run in linear time with respect to the length of the input string. You may use hashing under the assumption that unit space data can be hashed in unit time.

Use hashed value numbering. Create a map valnum(obj) that maps each object obj to a unique number between 1 and the number of distinct values occurring in the syntax tree, so that valnum(n1) = valnum(n2) iff n1 and n2 represent identical values. We simultaneously create a map valnodes(v) from each value number v to the set of nodes in the tree that have value number v. Modify the syntax tree so that the top-level attribute/value pairs of the same record r are logical siblings with r as their common parent. This effects how the height of a node is defined.

i. **Assign successive numbers valnum(i) starting from 1 for each distinct integer or attribute i occurring in the syntax tree, and associate valnum(i) with each node that has integer i. Next, pass through each set valnode(v) from v=1 to the highest assigned value number, and sort all attribute/value pairs uniformly for each record, according to ascending attribute value number. At the same time, we can prune duplicate attributes so that each attribute is followed by one or more nodes representing values associated with these duplicates. Such nodes must eventually have the same value numbers or the data is invalid.**

ii. **For height h = 1,2,..., assign value numbers to all nodes n at height h as follows: if n represents a record with attribute/pairs ai and vi for i=1,...,k, sorted according to attribute value number, make sure that all values associated with the same attribute have the same value number, and prune all such duplicates. Let obj be the k+1-tuple**

[record,valnum(a1),valnum(v1),...,valnum(ak),valnum(vk)].

If valnum(obj) is undefined, assign a next successive value number to valnum(obj). Associate valnum(obj) with node n.

6. Consider an imperative programming language with procedures, function style procedure calls, return statements, and assignments, but no nested procedure definitions; i.e., procedures can only be declared within the main program, and not within procedures. Assume also that procedure names cannot be passed arguments to procedures, and cannot be returned as results. Finally, assume call-by-value semantics.

   Show how to compute a program call-graph *cgraph(caller, called)*, where procedure *caller* contains a call to procedure *called*. When the procedure invocation is not within any procedure, then the caller is `main`. Use *cgraph* and any other relevant relations to perform procedure integration; i.e., correctly replace the procedure call by the body of the called procedure. Your algorithm should follow the rule: if procedure q is called by procedure p, integrate q into p if and only if the call graph contains no path from p to q.

   Let proc_call(p) be the list of procedure calls contained within procedure p or main. We can compute cgraph as follows:

   ```
   cgraph := {}
   for [p,pcalls] in proc_call loop
     for q in pcalls loop
       cgraph{p} with:= q
     end loop
   end loop
   ```

   Next, perform a strong component decomposition of cgraph. Each strong component contains a set of procedures with potential mutual recursion. Consider the dag representation of the strong components. Let the height of each component c be the length (in number of edges) of the longest path from c to a leaf in the dag. Observe that leaf components have height 0. For each height h = 1,..., and each procedure p contained in some component at height h, modify p as follows:

   - **(integration) For every call q(a1,...,ak) that occurs in p to a procedure q, if q belongs to a component of height less than h and if f1,...,fk are the formal parameters of q, then form a new code block q' by replacing all occurrences of fi by ai within the body of q, and replacing each command 'return R' by assignment t := R, where t is a new compiler generated variable.**
   - **If C is an innermost command containing the call q(a1,...,ak), then insert the statements of q' just before C, and replace call q(a1,...,ak) within C by t.**