Computer Science 15–312: Programming Languages

Final Examination

(With Sample Solutions)

May 9, 2000

Name

Name: ______________________________

Andrew ID: ______________________________

Instructions

- This is an open-book, open-notes examination. You have **3** hours to complete this examination.
- There are **16** pages in this examination, including two worksheets.
- This examination consists of **6** questions worth a total of **150** points.
- Read each question completely before attempting to solve any part.
- Write your answers legibly *in the space provided* on the examination sheet. If you use the back of a sheet, indicate clearly that you have done so on the front.
- Write your name and Andrew ID on this page and on the top of each successive page in the space provided.
- Two worksheets are attached to the end of this examination for your use.

Grading

Question	1	2	3	4	5	6	Total
Score							
Maximum							

1 Short Answer Questions

25 Points

Answer the following questions in the space provided, or circle the correct answer when a choice is given. points.

1. [9 points] Fix a universe $\mathcal{U}$. A (finite or infinite) *sequence* of elements of $\mathcal{U}$ is a partial function $\sigma : N \rightharpoonup \mathcal{U}$ such that either $\sigma(n)$ is defined (written $\sigma(n)\downarrow$) for all $n \in N$, or there exists $n \in N$ such that $\sigma(m)$ is defined for all $m < n$ and undefined for all $m \geq n$.

   Consider the set T inductively defined by the following inference rule:

   $$\frac{\forall n \in \mathbb{N}\ \sigma(n)\downarrow \textit{ implies } \sigma(n)\ \mathsf{tree}}{\mathit{Node}(\sigma)\ \mathsf{tree}}$$

   The elements of this set may be thought of as trees each of whose nodes has either a finite or an infinite number of children.

   (a) What is the base case of this inductive definition?

   (b) True or false: the set T contains finitely many elements.

   (c) True or false: every tree in T has a finite height given by the equation

   $$\mathit{height}(\mathit{Node}(\sigma)) = 1 + \max_{i \in N,\ \sigma(i)\downarrow}(\mathit{height}(\sigma(i)).$$

2. [6 points] The following questions pertain to a *call-by-value*, *dynamically typed* language:

   (a) True or false: in such a language it is possible to dispatch at run-time on the static type of an expression.

   (b) In such a language the following expression (circle one) incurs a compile-time error, incurs a run-time error, or evaluates normally:

   ```
   (if 3>=1+2 then 1 else "1") + 7
   ```

   (c) In such a language the following expression (circle one) incurs a compile-time error, incurs a run-time error, or evaluates normally:

   ```
   fst (3+5, 3+"5") + 7
   ```

3. [10 points] The following questions pertain to classes and types in Java:

   (a) True or false: Every instance has a unique type at compile time.

   (b) True or false: Every instance has a unique type at run time.

   (c) True or false: Every class extends at most one other class.

   (d) True or false: Every class implements at most one interface.

   (e) True or false: Every type has at most one *smallest* supertype.

1. (a) $Node(\bot)$, where $\bot$ is the completely-undefined function $N \to \mathcal{U}$.
   (b) False.
   (c) False. The tree with infinitely many children with heights 0, 1, 2, *etc.*has no finite height.
2. (a) False. Only the tag, not the full type.
   (b) Evaluates normally.
   (c) Incurs a run-time error.
3. (a) False. You can promote objects up the subtype hierarchy.
   (b) True. The class of an instance determines the methods that will execute.
   (c) True. Java has only single inheritance.
   (d) False. Java admits multiple "inheritance" of interfaces.
   (e) False. Due to interface types.

2 Operational Semantics 25 Points

In this question we investigate *delayed* evaluation as a primitive construct of MinML. We introduce a type $\tau\,\texttt{delayed}$ for delayed computations, with constructor $\texttt{delay}(e)$ and destructor $\texttt{force}(e)$. The former suspends the computation of e, while the latter 'thaws' the expression.

Here are the syntax and the typing rules for these constructs:

$$\begin{array}{rcll} \textit{Types} & \tau & ::= & \ldots \mid \tau\,\texttt{delayed} \\ \textit{Expressions} & e & ::= & \ldots \mid \texttt{delay}(e) \mid \texttt{force}(e) \\ \textit{Values} & v & ::= & \ldots \mid \texttt{delay}(e) \end{array}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{delay}(e) : \tau\,\texttt{delayed}} \qquad \frac{\Gamma \vdash e : \tau\,\texttt{delayed}}{\Gamma \vdash \texttt{force}(e) : \tau}$$

2.1 Operational semantics without memoization 10 Points

Give the SOS evaluation rules for these constructs. You should require no more than two rules in your solution.

2.2 Memoization 15 Points

Memoization is a programming technique for caching the results of previous computations so that they can be quickly retrieved without repeated effort. Memoization is fundamental to the implementation of lazy data structures. The idea is that a delayed computation is performed at most once. If it is never forced, then the delayed computation is never performed at all. If it is ever forced, then the *result* of forcing that computation is stored in a memo pad so that if it is forced again, the previous result is returned immediately, without repeating the work that was done previously. For example:

```
let d = delay(2*2*2) in force(d)+force(d) end
```

The second time we force the expression, the cached result is retrieved and returned; the multiplications are only performed once.

You are to define an abstract machine semantics for memoized delayed evaluation. Give only the rules for the constructs described above. You need not take account of exceptions or run-time errors; consider only "pure" MinML extended with delayed evaluation.

You will need to introduce one additional form of expression. The state of the abstract machine will consist of two parts, which you are to define. You are to define SOS rules for the abstract machine. **Important**: it is *essential* that your solution capture the "evaluate at most once" semantics of memoized delayed evaluation!

- The set of expressions and values

- The state of the machine

- The SOS rules

2.3 Solution

1.

$$\frac{e \mapsto e'}{\texttt{force}(e) \mapsto \texttt{force}(e')} \qquad \frac{}{\texttt{force}(\texttt{delay}(e)) \mapsto e}$$

2.
   - Introduce an infinite set of memo locations $d \in \mathcal{D}$.

$$\begin{array}{rcll} \textit{Expressions} & e & ::= & \ldots \mid \texttt{delay}(e) \mid \texttt{force}(e) \mid d \\ \textit{Values} & v & ::= & \ldots \mid \texttt{delay}(e) \mid d \end{array}$$

   - Let μ be a function from memo locations to delayed expressions, that is $\mu : \mathcal{D} \to \{delay(e)\}_{e \in exp}$: then the machine is the pair (μ, e).
   -

$$\frac{d \notin \mathrm{dom}(\mu)}{(\mu, \texttt{delay}(e)) \mapsto (\mu[d{=}\texttt{delay}(e)], d)}$$

$$\frac{(\mu, e) \mapsto^* (\mu', v)}{(\mu[d = \texttt{delay}(e)], \texttt{force}(d)) \mapsto (\mu'[d = \texttt{delay}(v)], v)}$$

$$\frac{(\mu, e) \mapsto (\mu', e')}{(\mu, \texttt{force}(e)) \mapsto (\mu', \texttt{force}(e'))}$$

3 Abstract Machines 25 Points

In this question we will consider a programming language FIFTH that is similar to the cult language FORTH. The dynamic semantics of FIFTH are defined in terms of an abstract machine. The state of the abstract machine consists of three stacks, the *program stack*, the *data stack*, and the *return stack*, and a *dictionary*. A FIFTH program is a stack of *words*, each of which is interpreted as a command to change the state of the FIFTH machine.

FIFTH programs are written in "reverse polish notation" in which the arguments to an operation are popped from the top of the data stack and replaced by the result of applying the operation. For example, the ML expression `(4 + 3) * 8` would be written as `4 3 + 8 *` in FIFTH, with the interpretation that the integers 4 and 3 are pushed onto the data stack, then are popped and replaced by their sum, then 8 is pushed, and the top two elements are popped and their product is pushed onto the stack. The sequence `4 3 + 8 *` is to be read as a stack of words, with the top of the stack at the left and the bottom at the right.

FIFTH programs are made up of the following words with the following interpretations as commands. Execution of an instruction causes that instruction to be popped from the program stack.

- Integer constants such as 0, 1, 2, ... Push the value of the constant onto the data stack.
- Addition, `+`. Pop the top two elements of the data stack, then push their sum onto the data stack.
- `DUP`. Duplicate the top element of the data stack.
- `DROP`. Delete the top element of the data stack.
- `SWAP`. Swap the top two elements of the data stack.
- '`.`'. Print the top element of the stack.
- '`:`'. Introduces a definition into the dictionary. The following word is the word to be defined; its definition is the sequence of words following the defined word and up to the next '`;`', regarded as a program stack.
- `IF`. Compare the top element of the data stack with zero, popping that element. If it is zero, the top element of *program stack* is popped. Otherwise, the top two elements of the *program stack* are swapped, and the top element is popped.

Some examples (from a FIFTH interpreter written in SML):

```
ok 4 5 + 8 9 + * .
153
ok : FACT1 DUP 1 - FACT * ;
ok : FACT2 DROP 1 ;
ok : FACT DUP IF FACT1 FACT2 ;
ok 5 FACT .
120
ok 6 FACT .
720
ok : FACT1 DROP 2 ;
ok 5 FACT .
2
ok 6 FACT .
2
```

(`ok` is the interactive prompt)

Notice that FIFTH is dynamically scoped in that any reference to a defined word looks up the most recent definition of the word in the dictionary and transfers execution to the code defined there. Changing the definition of `FACT1` changes the behavior of `FACT`!

The problem is to define a dynamic semantics for the FIFTH machine. The state of the FIFTH machine is a 4-tuple (d, D, R, P), where d is a dictionary, D is a data stack, R is a return stack, and P is a program stack. The sets of data, return, and program stacks are inductively defined as follows:

$$\frac{}{\bullet\ \textsf{progstack}} \qquad \frac{w\ \textsf{word} \quad P\ \textsf{progstack}}{w \triangleright P\ \textsf{progstack}}$$

$$\frac{}{\bullet\ \textsf{datastack}} \qquad \frac{n\ \textsf{int} \quad D\ \textsf{datastack}}{n \triangleright D\ \textsf{datastack}}$$

$$\frac{}{\bullet\ \textsf{retstack}} \qquad \frac{p\ \textsf{progstack} \quad R\ \textsf{retstack}}{p \triangleright R\ \textsf{retstack}}$$

Here n int is true for all integers and w word is true for all the pre-defined words (as described above) and any identifier.

The dictionary contains a mapping from words to programs (*i.e.*, stacks of words). You may use these two operations on dictionaries: $(word, stack) + d$, where d is a dictionary, gives a dictionary containing all definitions in d, plus (possibly superseding a previous definition) a mapping from the *word* to the *stack*. $d(word)$ returns as a program stack the definition of *word* (or causes an error if *word* is not defined).

1. [5 points]. When is a FORTH-machine state `final`?

2. [20 points]. Give evaluation rules for: $w(n)$, `IF`, `:`, `DUP`, `SWAP` and for user-defined words. Also give an evaluation rule for when the program-stack is empty. You may use $x \triangleright S$ to indicate a

stack with x as its top element and S as the rest of the stack (or, more generally, $x \triangleright y \ldots \triangleright S$).

3. [5 points extra credit]. Give a revised version of the evaluation rules that must be changed if our FIFTH machine is to be *tail-recursive* (in the usual sense of the term–state is only saved for a return when there are pending operations).

1. Answer 1. Any state $(d, D, \bullet, \bullet)$ is *final*. (Alternatively, the student may require D to be empty, but this is not required (or true in real FIFTH)).

2. Answer 2.

$$\frac{}{(d, D, R, w(n) \triangleright P) \mapsto (d, n \triangleright D, R, P)}$$

$$\frac{n = 0}{(d, n \triangleright D, R, \texttt{IF} \triangleright then \triangleright else \triangleright P) \mapsto (d, D, R, else \triangleright P)}$$

$$\frac{n \neq 0}{(d, n \triangleright D, R, \texttt{IF} \triangleright then \triangleright else \triangleright P) \mapsto (d, D, R, then \triangleright P)}$$

$$\frac{}{(d, D, R, \ : \triangleright word \triangleright \ldots \triangleright ; \triangleright P) \mapsto (d + (word, \ldots), D, R, P)}$$

$$\frac{}{(d, n \triangleright D, R, \texttt{DUP} \triangleright P) \mapsto (d, n \triangleright n \triangleright D, R, P)}$$

$$\frac{}{(d, n_1 \triangleright n_2 \triangleright D, R, \texttt{SWAP} \triangleright P) \mapsto (d, n_2 \triangleright n_1 \triangleright D, R, P)}$$

$$\frac{}{(d, D, R, word \triangleright P) \mapsto (d, D, P \triangleright R, d(word))}$$

$$\frac{}{(d, D, P \triangleright R, \bullet) \mapsto (d, D, R, P)}$$

3. Extra Credit Answer.

$$\frac{P \neq \bullet}{(d, D, R, word \triangleright P) \mapsto (d, D, P \triangleright R, d(word))}$$

$$\frac{P = \bullet}{(d, D, R, word \triangleright P) \mapsto (d, D, R, d(word))}$$

4 Polymorphic programming $\boxed{\textbf{25 Points}}$

We will define `div`$: N \to N \to N$ such that

$$\texttt{div}\ \overline{m}\ \overline{n} = (\overline{\tfrac{m}{n}},\ \overline{m \bmod n})$$

With $N := \forall t(t \to (t \to t) \to t)$ and $B := \forall t(t \to t \to t)$, we assume that the following functions are available.

$\texttt{succ}\ \overline{n} = \overline{n+1}$
$\texttt{pred}\ \overline{n} = \overline{n-1}\ ,\ n \neq 0$
$\texttt{pred}\ \overline{0} = \overline{0}$
$\texttt{and}\ \overline{b_1}\ \overline{b_2} = \overline{b_1 \wedge b_2}$

1. (5 Points) Define `isOne`$: N \to B$ such that

   $\texttt{isOne}\ \overline{1} = \texttt{true}$
   $\texttt{isOne}\ \overline{n} = \texttt{false}\ ,\ n \neq 1$

   You may introduce subsidiary functions if you want to make your solution more readable.

   Hint:

   $$\overline{x} = \overline{1} \iff \overline{x} \neq \overline{0} \wedge \texttt{pred}\ \overline{x} = \overline{0}$$

2. (10 Points) Define `isEqual`$: N \to N \to B$ such that

   $\texttt{isEqual}\ \overline{x}\ \overline{x} = \texttt{true}$
   $\texttt{isEqual}\ \overline{x}\ \overline{y} = \texttt{false}\ ,\ x \neq y$

   You can use `isOne` defined above.

3. (10 Points) Define `div`$: N \to N \to N * N$ such that

$$\texttt{div}\ \overline{m}\ \overline{n}\ \texttt{=}\ (\overline{\tfrac{m}{n}}\texttt{,}\ \overline{m \bmod n})$$
$$\texttt{div}\ \overline{0}\ \overline{0}\ \texttt{=}\ (\overline{1}\texttt{,}\ \overline{0})$$

You can use `isEqual` defined above.

Hint: use the idea used in defining function `pred` in the homework.

1. We introduce two subsidiary functions `isZero`: $N \to B$ and `isNonzero`: $N \to B$:

```
isZero = fun x:N is x[B] (fun b:B is false end) true end
isNonzero = fun x:N is x[B] (fun b:B is true end) false end
```

Then,

```
isOne = fun x:N in and (isNonzero x) (isZero (pred x)) end
```

2.
```
isEqual = fun x:N is fun y:N is isOne (y[N] pred (succ x)) end end
```

3.
```
div2 = fun m:N is fun n:N is
  m[N * N]
  (fun (q, r):N*N is (isEqual r (pred n))[N*N] (succ p, 0̄) (q, succ r))
  (0̄, 0̄)
  end end
```

5 Subtyping and Inheritance **25 Points**

A key aspect of the Java design is the principle that if a class A extends a class B, then (the instance type of) A is a subtype of (the instance type of) B — subclassing implies subtyping. By the subsumption principle an instance a of class A may be provided in any context where an instance of type B is required.

1. [10 Points] To ensure that inheritance implies subtyping we must impose restrictions on method override. In FJ, as in Java, to override a method m defined in the superclass, the subclass method must have the same type. This leads to the following overriding rule in FJ:[1]

$$\frac{\begin{array}{c} T(A) = \texttt{class}\,A\,\texttt{extends}\,B\,\{\ \ldots\,;\ \ldots\ \} \\ \mathsf{type}(m,B) = B_1 \to B_0 \quad x{:}B_1, \texttt{this}{:}A \vdash e_0 : B_0 \end{array}}{B_0\,m\,(B_1\,x)\,\{\,\texttt{return}\,e_0\,;\,\}\ \mathsf{ok\ in}\ A}$$

The rule says that if the method m is to be overriden in subclass A of superclass B, its argument and result type in the subclass A must be the same as that in B. However, the type of `this` is assumed to be A, the subclass type, so that the method body may call other subclass methods.

This rule is overly restrictive — it is not necessary to require that the argument and result types of the method in the subclass agree precisely with those of the same method in the superclass. Fill in the following revised rule with the *most general* subtyping conditions that may be imposed that are consistent with the principle that subclassing implies subtyping:

$$\frac{\begin{array}{c} T(A) = \texttt{class}\,A\,\texttt{extends}\,B\,\{\ \ldots\,;\ \ldots\ \} \\ \mathsf{type}(m,B) = B_1 \to B_0 \quad x{:}A_1, \texttt{this}{:}A \vdash e_0 : A_0 \\ \boxed{} \end{array}}{A_0\,m\,(A_1\,x)\,\{\,\texttt{return}\,e_0\,;\,\}\ \mathsf{ok\ in}\ A}$$

2. [15 Points]

   It has been proposed to extend Java so that classes may contain *type declarations* similar to those found in ML. A type declaration has the form `type t =` T, where T is a Java type expression. A subclass may override a type declaration by providing a declaration of the form `type t =` U, where U is a subtype of T according to the rules of subtyping in Java.

   Consider the following example:

[1] For the sake of simplicity, we assume that methods have only one argument.

```
class C {
  C F = null;
}
class B {
  type t = Object
  t M (t x) { return x; }
}
class A extends B {
  type t = C
  t M (t x) { return x.F; }
}
```

These declarations are well-formed according to the rules of Java, extended with type declarations, since the class type `C` is a subtype of the type `Object`. The method `M` defined in class `B` is clearly well-formed. The method `M` defined in the subclass `A` is also well-formed since it returns an object of type `C`.

Show that under this proposed extension to Java subclassing does not imply subtyping by filling in the following template:

```
class Null {            // also a subclass of Object
; no fields or methods
}

class Ctx {
  Object MyMeth (B y) {
    return y.M(                    );
  }

  Object Crash = MyMeth (                    );
}
```

The method `MyMeth` provides a context in which an instance of class `B` is expected. The call to `MyMeth` provides a context in which an argument must be provided. Assuming that `A` is a subtype of `B`, fill in the blanks in a type-correct manner so as to incur a run-time type error. Conclude that subclassing does not imply subtyping in this extension to the language.

1.

$$\frac{\begin{array}{c} T(A) = \texttt{class}\,A\,\texttt{extends}\,B\,\{\ \ldots;\ \ldots\ \} \\ \mathsf{type}(m,B) = B_1 \to B_0 \quad x{:}A_1, \texttt{this}{:}A \vdash e_0 : A_0 \\ B_1 \texttt{<:} A_1 \qquad A_0 \texttt{<:} B_0 \end{array}}{A_0\, m\,(A_1\, x)\,\{\,\texttt{return}\, e_0\,;\,\}\ \mathsf{ok\ in}\ A}$$

2.

```
class Null {
; no fields or methods
}
class Ctx {
  Object MyMeth (B y) {
    return y.M(new Null());
  }
  Object Crash = MyMeth (new A());
}
```

6 Futures in CML

25 Points

A *future* is a mechanism for performing evaluation of an expression in parallel with another program. The idea is that in some situations evaluation of an expression can be started in advance of the point at which the result is needed. When this is the case the program creates a future, a data structure representing the future value of that expression. When the future is created, a thread is forked to commence evaluation of that expression. When the value of the future is required, the program *touches* that future to retrieve it. If the parallel evaluation has completed, the value is returned; otherwise the program blocks until the computation of the future completes. An interesting complication is that if the future expression raises an exception, the same exception must be raised when the future is touched by the program.

Your task is to implement the following signature of futures using only the *synchronous* `CML` primitives:

```
signature FUTURE = sig
  type 'a future
  val future : (unit -> 'a) -> 'a future
  val touch : 'a future -> 'a
end
```

The function `future` takes a `unit -> 'a` function, spawns a thread to compute the value of the function applied to `()`, and returns a *future* representing the future result of that evaluation. Calling `touch` blocks the caller until the result of the future is available. Touching a future should always yield the same result, *without* re-evaluating the original function. **You may *not* use reference cells in your implementation**. (You may find it useful to recall the implementation of reference cells using threads given in class.)

If an exception is raised during the execution of the evaluation, the proper behavior is to have *touching* the value re-raise the exception raised in the spawned thread. Otherwise, the raising of the exception will terminate the spawned thread and any attempt to touch the value will block forever (in fact, in practice, it will terminate the thread attempting to claim the promise — can you see why?) The exception should be re-raised every time the future is touched.

As an example of the use of futures, we will consider computing Ulam's function (often called the $3n + 1$ function):

```
local
  fun ulam' (4, i, c) = i
    | ulam' (n, i, c) =
    if (i > c) then
      raise Cutoff
    else
      if (n mod 2) = 0 then
        ulam' (n div 2, i + 1, c)
      else
        ulam' (3 * n + 1, i + 1, c)
in
  fun ulam (n, c) = ulam' (n, 0, c)
end
```

Or, in mathematical notation,

```
ulam(n) = n/2  if n is even
          3n+1 if n is odd
```

Iteratively computing Ulam's function yields sequences such as: 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ... It is believed (but not known) that all such sequences result in the 4, 2, 1 loop seen above. We define a function that takes a number, a "cut off" (a maximum length for the sequence before terminating), and returns the number of terms in the Ulam sequence before reaching the 4 and starting the 4, 2, 1 loop.

Using futures we can write code that will quickly generate an array of future values of the `ulam` function. We then use a `probe` function to examine the values (in any order, possibly directed by user input or the result of other, unpredictable calculations). Probing a number that causes the `Cutoff` exception to be raised (or, perhaps, an `Overflow`) will re-raise that exception every time the value is probed.

```
(* The type is: int * int -> unit -> int*)
fun ulamDelayed (i, c) () = ulam (i, c)
(* The type is: int * int -> int future array *)
fun ulamGen (n, c) =
  Array.tabulate (n, fn i => future (ulamDelayed (i, c)))
(* The type is: 'a future array * int -> 'a *)
fun probe (a, i) = touch (Array.sub (a, i))
```

(`Array.tabulate` takes an integer, n and a function, f and generates an array such that $A_i = f(i)$ for $i = 0 \ldots n-1$). The benefit of this approach is clear if we know that we will eventually want all or most of the values, but we cannot predict in what order we will need them–our computation is now demand-driven.

1. Part 1 (5 points). Define the type future and any other datatypes you will need (remember the requirement stated above concerning exceptions):

   ```
   type future =
   ```

2. Part 2 (15 points). Implement `future`, using only the **CML** primitives `spawn`, `channel`, `send` and `recv`. Do not use mutable storage (`ref` cells).

```
fun future f =
  let

    fun



  in




  end
```

3. Part 3 (5 points). Implement `touch`.

```
fun touch f =
```

1. Answer 1.

```
type 'a future = unit -> 'a
datatype 'a return = FRESULT of 'a | FEXN of exn
```

2. Answer 2.

```
fun future f =
  let
    val rc = channel ()
    fun spin result =
      (send (rc, result);
       spin result)
  in
    spawn (fn () =>
           spin ((FRESULT (f ())) handle e => FEXN e));
    (fn () =>
     case (recv rc) of
       FRESULT x => x
     | FEXN e => raise e)
  end
```

3. Answer 3.

```
fun touch f = f ()
```