Programming Languages (CS 328)
Computer Science and Engineering
Indian Institute of Technology Bombay

Midterm examination
2000-02-25
(Open notes)

The exam is graded out of 20 points. It is possible to score more than 20 points. The extra credit question is not necessarily more difficult compared to other questions. You are however advised to start on the extra credit question only after you have completed answering the other questions to your satisfaction. It is to your benefit to show the steps to your answer: even if the answer is wrong you may get partial credit. However, manage your time carefully, guided by the number of points for each question. In particular you need not write down unimportant steps in the evaluation. Answers are in a different color.

1. (3 points) Recall the definition of Church numerals as

$$\bar{n} \equiv (\lambda\ \mathtt{f}\ (\lambda\ \mathtt{a}\ (\mathtt{f}\ (\mathtt{f}\ \ldots(\mathtt{f}\ \mathtt{a})\ldots)))), \tag{1}$$

where `f` is applied to `a` n times where n corresponds to the Church numeral $\bar{n}$. With this definition we designed λ-expressions for `ZERO?`, `INC`, `DEC`, `SUM`, and `TIMES`.

Design `EXP` $\equiv$(λ `n` ____) which accepts a Church numeral $\bar{n}$ as argument and returns a Church numeral corresponding to $\overline{2^n}$. You may use any macro defined in class, not limited to the above list. Do not use recursion or the Y combinator.

There are many possible variants such as

$$\begin{aligned} \mathtt{EXP} &\equiv (\lambda\ \mathtt{n}\ ((\mathtt{n}\ (\mathtt{TIMES}\ \bar{2}))\ \bar{1})) \\ \text{and} \quad \mathtt{EXP} &\equiv (\lambda\ \mathtt{n}\ ((\mathtt{n}\ \bar{2})\ )). \end{aligned}$$

It takes some work to show that the latter works.

2. (4 points) Design a macro `HALF` $\equiv$(λ `n` ____) which accepts a Church numeral $\bar{n}$ and returns $\overline{\lfloor n/2 \rfloor}$. You may use any macro defined in class, not limited to the above list. Do not use recursion or the Y combinator.

Many variants have been suggested and are correct, one possibility is:

$$\begin{aligned} \mathtt{STEP} &\equiv (\lambda\ \mathtt{p}\ (\mathtt{IF}\ (\mathtt{RIGHT}\ \mathtt{p})\ (\mathtt{PAIR}\ (\mathtt{INC}\ (\mathtt{LEFT}\ \mathtt{p}))\ \mathtt{FALSE})\ (\mathtt{PAIR}\ (\mathtt{LEFT}\ \mathtt{p})\ \mathtt{TRUE}))) \\ \mathtt{HALF} &\equiv (\lambda\ \mathtt{n}\ (\mathtt{LEFT}\ (\mathtt{n}\ \mathtt{STEP}\ (\mathtt{PAIR}\ \bar{0}\ \mathtt{FALSE})))) \end{aligned}$$

Anything that is correctly written according to λ-grammar and works gets full credit. If the answer is verbal but correct, we will give 3 points. Near-correct solutions with correct idea will also get partial credit.

3. (Extra credit: 3 points) The standard definition of Church numerals is analogous to *unary* encoding: the syntactic complexity (number of nodes in the abstract λ-syntax tree) of $\bar{n}$ is $\Omega(n)$. Design an alternative representation for numerals which has only $O(\log n)$ syntactic complexity, which will make the representation as efficient as *binary* encoding. You may use any macro defined in class, not limited to the above list. Do not use recursion or the Y combinator. (Hint: solving questions 1 and 2 first may help you.)

Once you can create a `PAIR`, you can create a list, in which you can store the bits of n in binary encoding. This is not what we were looking for. In the old representation, to write down the literal numeral $\bar{n}$ would cost you $\Omega(n)$ space. It would be a procedure with two curried arguments, f and a and a body that applied f to a n times, where the cascading applications would be explicit in the body of (1). The question asked for an alternative representation with the *same* signature (λ `f` (λ `a` ...)) and the *same* semantics as the old Church numerals, but a new body which takes only $O(\log n)$ space to write down. We convert the unary body into a binary body using the following two equalities:

$$\begin{aligned} (f^{2m}\ a) &= (f^m\ (f^m\ a)) \\ &= (\bar{2}\ (\lambda\ a'\ (f^m\ a'))\ a), \quad \text{and} \qquad (2) \\ (f^{2m+1}\ a) &= (f^{2m}\ (f\ a)), \qquad (3) \end{aligned}$$

where each application of rule (3) enables an application of rule (2) which cuts down the remaining work by a factor of 2; therefore in $O(\log n)$ steps we will be done converting $\bar{n}$. For example,

$$\begin{aligned} (f^5\ a) &= (f\ (f\ (f\ (f\ (f\ a))))) \\ &= (\underline{f^4}\ (f\ a)) \\ &= (\bar{2}\ (\lambda\ a_1\ \underline{(f^2\ a_1)})\ (f\ a)) \\ &= (\bar{2}\ (\lambda\ a_1\ (\bar{2}\ (\lambda\ a_2\ (f\ a_2))\ a_1))\ (f\ a)) \end{aligned}$$

One may protest this is hardly a 'simplification,' but it *is* for large values of n.

4. (3 points) Evaluate the following expression in dynamically scoped pass-by-value FL. Note that unlike FLK which only has a single parameter `lambda` and an explicit `call` keyword, FL does not need the `call` keyword and has a multi-parameter `lambda` form in which the formal arguments are specified in a (possibly empty) list. A procedure `proc` with and empty argument list is applied as `(proc)`.

```
(let ((make-sub (lambda (x) (lambda (n) (- n x))))
      (apply-sub (lambda (f x) (f x))))
    (let ((x 1))
        (apply-sub (make-sub 20) 300) ) )
```

The expression evaluates to zero. Keep in mind that `(make-sub 20)` in effect evaluates to `(lambda (n) (- n x))` where the binding `x` $\rightarrow$ 20 has been lost. Next we enter `apply-sub`, masking the `x` $\rightarrow$ 1 binding with the new `x` $\rightarrow$ 300 binding. Next we create a binding `n` $\rightarrow$ 300 as a result of evaluating `(f x)`, and end up evaluating `(- n x)` where both are bound to 300.

We decided to award no points for an answer of 280 or -280, as those answers fail to distinguish the basics of static and dynamic scoping. An answer of 299 shows some cognizance that the situation is more complex than static scoping, so we decided to award that one point.

5. (4 points) Draw a 2×2 table to show the result of evaluating the given FL expression under static and dynamic scoping and pass by name and value.

```
(let ((bomb (lambda () ((lambda (x) 3) (/ 1 0)))))
    (let ((f (lambda () (bomb))))
        (let ((bomb (lambda ()
                    .   ((lambda (x) 5)
                    .    ((lambda (x) (x x)) (lambda (x) (x x)))
                    .   )
                    ) ))
            (f) ) ) )
```

	Static scope	Dynamic scope
Pass by name	3	5
Pass by value	`error`	$\perp$

For static scope, the inner redefinition of `bomb` can be ignored. Pass by name is not affected by the division by zero; pass by value is. For dynamic scoping the inner definition of `bomb` takes effect, leading to an infinite loop for pass by value. There is one point per slot and we are inclined towards 0/1 scoring.

6. (6 points) Java Virtual Machines in browsers (supposedly) use a *sandbox* to protect its memory from getting corrupted by a buggy or malicious Java applet. We can model this in FLK! by adding the language construct `(sandbox` E`)`, whose informal semantics are to evaluate E w.r.t. the current store, but then *discard* all changes made to the store during the evaluation of E. Locations created within E cannot be referenced outside the `sandbox`: e.g., `(^ (sandbox (ref 1)))` results in an error. The `sandbox` construct can be nested. Study the two examples shown below.

```
(let ((c (ref 0))
    (begin (sandbox (:= c 2))
           (^ c)))
==> 0
```

```
(let ((c (ref 3)))
    (+ (^ c)
       (sandbox (begin (:= c 2) (^ c))) ) )
==> 5
```

Then write a denotational semantics for `sandbox` and verify the intended behavior for the above examples. In class, thanks to our functional notation, denotational semantics involving the store could make copies of the complete store liberally (by just mentioning s more than once in the body of $\mathcal{E}$). In a practical interpreter this would be too inefficient. You will get full credit only if you can economize the extent of store-copying to a bare minimum. In your answer give:

(a) The new design of *Store*

(b) New access and update functions for the new design of *Store*

(c) Evaluation functions for `(ref` E`)`, `(^`E`)`, `(:=` E_1 E_2`)`, and `(sandbox` E`)`.

(Hint: a significant redesign of the store data structure is needed.)

We add an argument $n \in \mathit{LevelNumber}$ to the $\mathcal{E}$ function to turn it into $\lambda u.\lambda s.\lambda n.$____. The first call to $\mathcal{E}$ is made with $n = 0$. The level number is increased by one whenever the interpreter enters into the scope of a `sandbox`.

Many changes are made to *Store*. *Store* used to be a map from *Location* to *Value* + *Error*. Think of the redesigned *Store* as a map keyed by *LevelNumber* × *Location*. The accessor 'fetch' now has signature *Store* × *LevelNumber* × *Location* → *Value* + *Error*. In words, fetch(s, n, l) looks for a key (n', l) in s such that $n' \leq n$, returning the value in the corresponding cell if any, otherwise returning an error. A new reference is created at the current level.

To support assignment, we write an accessor write(s, n, l, v). If key (n, l) is found in s, write merely modifies the value in the corresponding cell to v and returns. If a key (n', l) exists in s with $n' < n$, write creates a new key (n, l) mapped to a new cell with value v stored in it, and returns. If no key (n', l) is found, write returns an error.

When returning from the evaluation of a `sandbox`, another accessor erase(s, n) is called on the store, which deletes all keys of the form $(n, \cdot)$ and corresponding cells.

Evaluation will be based on the following criteria:

- No credit will be given for making a complete copy of the store, giving SOS similar to `(try...)` or just explaining how the examples work.
- The solution must roll back updates to the store while exiting the sandbox to get any credit.
- Points will be taken off for not handling nested sandboxes.
- Points will be taken off if cells created in a sandbox is visible from outside.
- Points will be taken off if cells already created outside a sandbox are *not* visible inside it.
- Points will be taken off if the entire store is designed as a stack of list, which will slow down location accesses even if no sandbox scope is in effect.
- One point is allocated for converting an informal but correct idea into a clean, formal specification.