cs173: Programming Languages

Final Exam

Fall 2000

The questions on this exam are worth 101 points.
Exam rules:

- Your responses to this exam are due by 10am on 2000-12-14.
- If you believe a question is underspecified, make a reasonable assumption, and document your claim and your work-around.
- The exam is open book with respect to the course text (Krishnamurthi and Felleisen) and course lecture notes (Tucker and Krishnamurthi, with guest appearances by Blaheta and Renieris). You may cite these by lecture date. You may not refer to any other sources related to the course between the time you start and finish the exam.
- You have three hours and 15 minutes to take this exam. Time begins when you start reading the exam questions. The first three hours must be a single, contiguous block. After you have completed this period, you must close your responses. At some later point, at your discretion (and before the responses are due), you may open your responses for one 15 minute period to edit or append to them.
- All code responses must use the dialect of Scheme employed in this course. You may not use side-effects or continuations unless expressly permitted to do so in the problem statement.
- Staple your solutions together. Put your name on every page.
- Solutions may be hand-written, but the burden of legibility rests on you.
- You may wish to respond to questions on a print-out of the exam itself. This is especially useful when you are asked to modify code, but don't wish to copy all of it. (In these cases, be especially wary of illegible writing and markings.) If you do pen your response on a copy of the exam, attach any extra sheets where they belong in concert with the questions, rather than at the end.
- You may not evaluate any of programs related to this exam on a computer.
- You must neither give assistance to any other course participant, nor receive assistance from anyone other than the course staff, on material under examination.

Brown University has an Academic Code that governs this exam and all our other transactions. I expect you, as students and scholars, to abide by it faithfully and fully.

Enjoy the exam!

Problem 1: Type Judgments [10 point(s)]

Consider the language

$$\begin{array}{rcll} L & ::= & N & \text{where } N \in \{1,\,2,\,3,\,\ldots\} \\ & | & x & \\ & | & (\lambda\ (x : \tau)\ L) & \\ & | & (L\ L) & \end{array}$$

with the types

$$\begin{array}{rcl} \tau & ::= & int \\ & | & (\tau \to \tau) \end{array}$$

and the type judgments

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash N : int}$$

$$\frac{\Gamma + \{x : \tau'\} \vdash L : \tau}{\Gamma \vdash (\lambda\ (x : \tau')\ L) : (\tau' \to \tau)} \qquad \frac{\Gamma \vdash L_f : (\tau' \to \tau) \quad \Gamma \vdash L_a : \tau'}{\Gamma \vdash (L_f\ L_a) : \tau}$$

We extend this language with an explicitly typed version of the **i-letrec** ("irreflexive letrec") construct from the first exam. This has the form

$$\begin{array}{l} (\textbf{i-letrec}\ ([var_1\ :\ tau_1\ =\ expr_1] \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad\quad [var_n\ :\ tau_n\ =\ expr_n]) \\ \quad body) \end{array}$$

where each $expr_1$ is required to be a function (and each tau_i must therefore be an "arrow" type). The above expression extends its inherited environment as follows:

- $\{var_1, ..., var_n\} - \{var_k\}$ are bound in $expr_k$
- $\{var_1, ..., var_n\}$ are bound in *body*

i-letrec evaluates each of $expr_1$ through $expr_n$ in order, binding each value to the corresponding var_i, then evaluates and returns the value of *body*.
Formulate a type judgment for **i-letrec**.

Problem 2: Overloading and Type Systems [10 point(s)]

The homework on overloading stated that one of the problems with overloading is that it's difficult to assign a type to an overloaded function. Let's explore this.

Suppose we overload the name + with two implementations: it accepts either two number's, returning a number (their sum), or two string's, returning a string (their concatenation). Here are two possible ways of typing it. In each case, justify your response briefly.

1. The type

   $$+ : \alpha \times \alpha \rightarrow \alpha$$

   adequately handles all the instances of + described above. Is this a reasonable type for + in our language?

2. Type researchers have worked on several interesting extensions to the type systems we've studied so far; one of them is to add *union* types. This extends the type language with

   $$\begin{aligned} \tau ::= & \ \ldots \\ & \mid \tau \cup \tau \end{aligned}$$

   so if a programmer gave a variable the type (number ∪ string), it means that variable can hold either a number or a string, but nothing else. Any value of type τ also has type $\tau \cup \tau'$ (for arbitrary other type τ'), and the language provides a **cases**-like construct to branch on the type of a value. In this type language, we can write

   $$+ : (\mathsf{number} \cup \mathsf{string}) \times (\mathsf{number} \cup \mathsf{string}) \rightarrow (\mathsf{number} \cup \mathsf{string})$$

   Is this a reasonable type for +?

Problem 3: CPS [15 point(s)]

Convert the following program into CPS form. You may assume the following to be primitive, i.e., they need not be transformed: all built-in Scheme primitives, and all procedures defined by the datatypes. Naturally, you should not CPS the **define-datatype** statements.

```
(define-datatype LabelTree LabelTree?
  [LLeaf (v number?)]
  [LLabel (name string?) (tree LabelTree?)]
  [LNode (lhs LabelTree?) (rhs LabelTree?)])

(define-datatype UnLabelTree UnLabelTree?
  [ULeaf (v symbol?)]
  [UNode (lhs UnLabelTree?) (rhs UnLabelTree?)])

;; symbolic-sign : number → symbol

(define (symbolic-sign v)
  (cond
   [(< v 0) 'negative]
   [(= v 0) 'zero]
   [(> v 0) 'positive]))

;; tree-transform : LabelTree × (number → symbol) → UnLabelTree

(define (tree-transform tree trans)
  (cases LabelTree tree
    [LLeaf (v) (ULeaf (trans v))]
    [LLabel (label subtree) (tree-transform subtree trans)]
    [LNode (lhs rhs)
     (UNode (tree-transform lhs trans)
            (tree-transform rhs trans))]))

;; this is a use of the above functions; transform this also

(tree-transform
 (LNode (LNode (LLeaf 2)
               (LLabel "jack" (LLeaf −1)))
        (LLabel "jill" (LNode (LNode (LLeaf 1)
                                     (LLeaf 4))
                              (LLeaf 0))))
 symbolic-sign)
```

Problem 4: Compilers and Run-Time Systems [15 point(s)]

Consider a language with arrays. A programmer might write

$A[i + j - k]$

This expression evaluates to the $(i+j-k)^{\text{th}}$ value in the array A. In a typical assembly language, this might compile to

load A **into** reg1 ;; loads address of A into reg1
add i **to** reg1 ;; adds value of i
add j **to** reg1 ;; adds value of j
sub k **from** reg1 ;; subtracts value of k
load reg1 **into** reg2 ;; loads value at reg1 into reg2

so reg2 contains the desired array value.
Suppose that $i + j - k$ always evaluates to a legal subscript for the array A, so you need not worry about safety problems.

1. In an AFunExp program, can the collector *possibly* be invoked while the program is executing in the midst of this sequence of instructions?
2. In some language (maybe AFunExp, or else some extension of it), assume the collector can possibly be invoked in the midst of this instruction sequence.
   (a) Can the above sequence of instructions interfere with the garbage collector's effectiveness in a non-copying collector? How?
   (b) Does anything change if it's a copying collector?

Problem 5: Copying Garbage Collection [15 point(s)]

Does *copy-location* in the stop-and-copy code handle circularity? Construct a scenario with a circular data structure by presenting the contents of the heap and of the entire root set. (You don't have to produce a program that creates the data.) Then justify your answer in terms of this root set and heap. If *copy-location* works, explain how. If it doesn't work, explain why not and sketch a solution.

Problem 6: Conservative Garbage Collection [10 point(s)]

Conservative garbage collectors are designed for use in *hostile* languages: ones whose definitions make it impossible to accurately identify roots and pointers. Yet conservative collectors are also applied to languages whose definitions make it possible to find all roots and pointers, e.g., Scheme or Java. Provide three distinct reasons for using conservative collectors in implementations of languages that are cooperative rather than hostile. (These reasons might not be independent—one may enhance the other—but they should largely be about distinct ideas.) Be precise and brief.

Problem 7: Copying Conservative Garbage Collection [5 point(s)]

We've seen that garbage collectors can compact the heap by copying values. A typical conservative collector cannot, however, copy values, because C and C++ allow programmers to temporarily hide pointers from the collector by casting, arithmetic and other techniques; because the collector cannot update pointers it does not see, when the pointers re-materialize, they may refer to the wrong data.

Suppose a conservative collector is linked to a C/C++ program generated by a compiler from an ML or Java source program. The compiler writer guarantees that the generated code does not hide pointers. He says the generated code adheres to C/C++ values where possible, e.g., numbers are represented by C numbers. He wants the author of the conservative collector to build a special-purpose copying mode: it would be conservative because the compiler does not generate tags for most data, but it can copy since it can find, and thus update, all the pointers.

On technical grounds, should the conservative collector's author agree to offer this special mode? If so, does it matter whether it's generational or a full-heap, semi-space stop-and-copy? If not, why not?

Problem 8: ML+ [20 point(s)]

A little-known computer corporation in Washington has gotten hooked on the power of ML, and want to use it for in-house development. In his glory days, the corporation's chairman added a few key memory reference routines to BASIC. He wants to repeat his success on ML.
Respecting the spirit of the language, he first defines a new basic type, address, and describes these operations in terms of that type. Since you have attended cs173, he hires you to add the type and these operations:

address_of : $\alpha \rightarrow$ address
peek : address $\times$ int $\rightarrow$ string
poke : address $\times$ string $\rightarrow$ address

address_of may be applied to any value implemented by a *Reference* (rather than an *Immediate*); the run-time system checks this before applying the operation. It returns the memory address of the argument value. **peek** (x, y) returns the y bytes starting at address x as a string. **poke** consumes a string and writes it into memory, returning the last memory address whose value it mutates.
You are given an interpreter (this chairman is also a fan of interpreters) with a tagged heap and copying garbage collector. For each of the following parts of the language, estimate how much effort you think is involved (an hour, a week, a month, a year, a PhD, impossible, etc) to (a) implement the changes at all and (b) additionally, do so in a way that preserves relevant properties of the original language (such as type soundness). If a modification is simple, explain in a sentence what it is. If, on the other hand, you think it's significant, write at most a paragraph or two explaining both why it's involved, and (to the best of your knowledge) what needs to be done.

1. parser
2. type inference engine
3. interpreter
4. garbage collector

Problem 9: Bonus Question [1 point(s)]

Name a State Troubador of Connecticut.

This space left free for doodling.