Comp 311: Sample Exam II

November 24, 2008

Name: ______________________________

Id #: ______________________________

Instructions

1. The examination is closed book. The type checking rules for (Implicitly) Polymorphic Jam are given on the first three pages of the exam as a reference.
2. Fill in the information above and the pledge below.
3. There are 7 problems on the exam worth a total of 110 points.
4. You have four hours to complete the exam. You must take the exam during a continuous four hour block plus an optional 10 minute break. Do not discuss the contents of the exam with anyone other than the instructor and teaching assistants between now and the due date for the exam.

Pledge:

Synopsis of Implicitly Polymorphic Jam

The syntax of (Implicitly) Polymorphic Jam is a restriction of the syntax of untyped Jam. Every legal Polymorphic Jam program is also a legal untyped Jam Program. But the converse is false, because there may not be a valid typing for a given untyped Jam program.

Abstract Syntax

The following grammar describes the abstract syntax of Polymorphic Jam. Each clause in the grammar corresponds directly to a node in the abstract syntax tree. The `let` construction has been limited to a single binding for the sake of notational simplicity. It is straightforward to generalize the rule to multiple bindings (with mutual recursion). Note that `let` is *recursive.*

$M ::= M\ \texttt{(}M \cdots M\texttt{)} \mid P\ \texttt{(}M \cdots M\texttt{)} \mid \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \mid \texttt{let}\ x\ \texttt{:=}\ M\ \texttt{in}\ M$
$\mid V$
$V ::= \texttt{map}\ x \cdots x\ \texttt{to}\ M \mid x \mid n \mid \texttt{true} \mid \texttt{false} \mid \texttt{null}$
$n ::= \texttt{1} \mid \texttt{2} \mid \ldots$
$P ::= \texttt{cons} \mid \texttt{first} \mid \texttt{rest} \mid \texttt{null?} \mid \texttt{cons?} \mid \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{*} \mid \texttt{=} \mid \texttt{<} \mid$
$\texttt{<=} \mid \texttt{<-} \mid \texttt{+} \mid \texttt{-} \mid \texttt{\~{}} \mid \quad \texttt{ref} \mid \texttt{!}$
$x ::=$ variable names

In the preceding grammar, unary and binary operators are treated exactly like primitive functions.

Monomorphic types in the language are defined by τ, below. Polymorphic types are defined by σ. The $\rightarrow$ corresponds to a function type, whose inputs are to the left of the arrow and whose output is to the right of the arrow.

$\sigma ::= \forall \alpha_1 \cdots \alpha_n .\, \tau$
$\tau ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \tau_1 \times \cdots \times \tau_n \rightarrow \tau \mid \alpha \mid \mathsf{list}\,\tau \mid \mathsf{ref}\,\tau$
$\alpha ::=$ type variable names

Type Checking Rules

In the following rules, the notation $\Gamma[x_1 : \tau_1, \ldots, x_n : \tau_n]$ means the $\Gamma \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$.

$$\Gamma \vdash \texttt{true} : \mathsf{bool} \qquad \Gamma \vdash \texttt{false} : \mathsf{bool} \qquad \Gamma \vdash n : \mathsf{int}$$

$$\frac{\Gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash M : \tau}{\Gamma \vdash \texttt{map}\ x_1 \ldots x_n\ \texttt{to}\ M : \tau_1 \times \cdots \times \tau_n \rightarrow \tau}[\textbf{abs}]$$

$$\frac{\Gamma \vdash M : \tau_1 \times \cdots \times \tau_n \rightarrow \tau \qquad \Gamma \vdash M_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash M_n : \tau_n}{\Gamma \vdash M\ \texttt{(}M_1 \cdots M_n\texttt{)} : \tau}[\textbf{app}]$$

$$\frac{\Gamma \vdash M_1 : \textsf{bool} \qquad \Gamma \vdash M_2 : \tau \qquad \Gamma \vdash M_3 : \tau}{\Gamma \vdash \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 : \tau}[\textbf{if}]$$

Note that there are two rules for `let` expressions. The [**letmono**] rule corresponds to the **let** rule of Typed Jam; it places no restriction on the form of the right-hand side M_1 of the `let` binding. The [**letpoly**] rule generalizes the free type variables (not occurring in the type environment Γ) in the type inferred for the right-hand-side of a `let` binding – provided that the right-hand-side M_1 is a *syntactic* value: a *polymorphic constant* like `null`, a `map` expression, or a variable. Syntactic values are expressions whose evaluation is trivial, excluding evaluations that allocate storage.

$$\Gamma[x : \tau] \vdash x : \tau \qquad\qquad \frac{\Gamma[x : \tau'] \vdash M_1 : \tau' \qquad \Gamma[x : \tau'] \vdash M_2 : \tau}{\Gamma \vdash \texttt{let}\ x\ \texttt{:=}\ M_1\texttt{;}\ \texttt{in}\ M_2 : \tau}[\textbf{letmono}]$$

$$\frac{\Gamma[x : \tau'] \vdash V : \tau' \qquad \Gamma[x : \text{CLOSE}(\tau', \Gamma)] \vdash M : \tau}{\Gamma \vdash \texttt{let}\ x\ \texttt{:=}\ V\texttt{;}\ \texttt{in}\ M : \tau}[\textbf{letpoly}]$$

$$\Gamma[x : \forall \alpha_1, \ldots, \alpha_n.\, \tau] \vdash x : \text{OPEN}(\forall \alpha_1, \ldots, \alpha_n.\, \tau,\ \tau_1, \ldots, \tau_n)$$

The functions OPEN and CLOSE are the keys to polymorphism. Here is how CLOSE is defined:

$$\text{CLOSE}(\tau, \Gamma) := \forall \{\text{FTV}(\tau) - \text{FTV}(\Gamma)\}.\, \tau$$

where FTV(α) means the "free type variables in the expression (or type environment) α".

When closing over a type, you must find all of the free variables in τ that are not free in any of the types in the environment Γ. Then, build a polymorphic type by quantifying τ over all of those type variables.

To open a polymorphic type

$$\forall \alpha_1, \ldots, \alpha_n.\, \tau,$$

substitute the chosen type terms $\tau_1, \ldots, \tau_n$ for the quantified type variables $\alpha_1, \ldots, \alpha_n$:

$$\text{OPEN}(\forall \alpha_1, \ldots, \alpha_n.\, \tau,\ \tau_1, \ldots, \tau_n) = \tau_{[\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n]}$$

which creates a monomorphic type from a polymorphic type. For example,

$$\text{OPEN}(\forall \alpha.\, \alpha \to \alpha, \tau) = \tau \to \tau$$

Types of Primitives

The following table gives types for all of the primitive functions and operators and the polymorphic constant `null`. Programs are type checked starting with a primitive type environment consisting of this table.

`null`	$\forall\alpha.\ \mathsf{list}\ \alpha$
`cons`	$\forall\alpha.\ \alpha \times \mathsf{list}\ \alpha \rightarrow \mathsf{list}\ \alpha$
`first`	$\forall\alpha.\ \mathsf{list}\ \alpha \rightarrow \alpha$
`rest`	$\forall\alpha.\ \mathsf{list}\ \alpha \rightarrow \mathsf{list}\ \alpha$
`cons?`	$\forall\alpha.\ \mathsf{list}\ \alpha \rightarrow \mathsf{bool}$
`null?`	$\forall\alpha.\ \mathsf{list}\ \alpha \rightarrow \mathsf{bool}$
`=`	$\forall\alpha.\ \alpha \times \alpha \rightarrow \mathsf{bool}$

`+`	$\mathsf{int} \times \mathsf{int} \rightarrow \mathsf{int}$
`-`	$\mathsf{int} \times \mathsf{int} \rightarrow \mathsf{int}$
`*`	$\mathsf{int} \times \mathsf{int} \rightarrow \mathsf{int}$
`/`	$\mathsf{int} \times \mathsf{int} \rightarrow \mathsf{int}$
`<`	$\mathsf{int} \times \mathsf{int} \rightarrow \mathsf{bool}$
`<=`	$\mathsf{int} \times \mathsf{int} \rightarrow \mathsf{bool}$
(unary) `-`	$\mathsf{int} \rightarrow \mathsf{int}$
(unary) `+`	$\mathsf{int} \rightarrow \mathsf{int}$
(unary) `~`	$\mathsf{bool} \rightarrow \mathsf{bool}$
`<-`	$\forall\alpha.\ \mathsf{ref}\ \alpha \times \alpha \rightarrow \mathsf{unit}$
`ref`	$\forall\alpha.\ \alpha \rightarrow \mathsf{ref}\ \alpha$
`!`	$\forall\alpha.\ \mathsf{ref}\ \alpha \rightarrow \alpha$

Typed Jam

The Typed Jam language used in Assignment 5 (absent the explicit type information embedded in program text) can be formalized as a subset of Polymorphic Jam. For the purposes of this test, Typed Jam is simply Polymorphic Jam less the **letpoly** inference rule which prevents it from inferring polymorphic types for program-defined functions.

Problem 1. [15 points]

(*i*) [5 points] Give a simple example of an untyped Jam expression (which is *not* a value) that is *not* typable in Polymorphic Jam, yet does not generate a run-time error when executed. Briefly but convincingly explain why.

The program `(map x to x(x)) (map x to x(x))`, commonly called Omega, is not typable in Polymorphic Jam yet does not generate a run-time error. It is not typable because the body of the function `(map x to x(x))` contains the self-application `x(x)` which is not typable. `x(x)` is not typable because `x` has a function type $\alpha \to \beta$ with an input type α that equals the function type $\alpha \to \beta$. But these two type are not unifiable because $\alpha \to \beta$ contains α. Circular bindings of type variables are not allowed in Polymorphic Jam.

The program does not generate a run-time error because `(map x to x(x)) (map x to x(x))` is not a value but directly ("in one step") reduces to itself generating a divergent computation.

(*ii*) [5 points] Give a simple example of an untyped Jam expression that is not typable in Typed Jam, but is typable in Polymorphic Jam. Briefly but convincingly explain why.

The program `let id = map x to x; in (id(id))(17)` is typable in Polymorphic Jam but not in Typed Jam because the `id` function is polymorphic. It is used with two different typings in the body of the `let`. Tbe inner occurrence of `id` has type `int` $\to$ `int` while the outer occurrence has type `(int` $\to$ `int)` $\to$ `(int` $\to$ `int)`.

(*iii*) [5 points] Assume that we extend Polymorphic Jam by dropping the "value restriction" on the right hand side of bindings in **letpoly** rule and add the block construct (definable as an expansion into **map** application) and the corresponding typing rule. Give a simple example of a program that is typable in extended Polymorphic Jam but generates a run-time *type* error (misinterpreting one type of data as another) when it is executed.

```
let fn := ref(map x to x);
in {
  fn <- map x to x+1;
  (!fn)(true);
}
```

The preceding program uses `fn` polymorphically: once as type `ref(int` $\to$ `int))`, so the assigment to `fn` is type correct, and once as type `ref(bool` $\to$ `bool`, so the application of `!fn` to true is type correct. But the assignment places a function of type `int` $\to$ `int` in the cell `fn`, which fails when it is applied to `true` because `!fn` tries to add `1` to its argument `x`.

There are no polymorphic values in Polymorphic Jam (or in ML/OCaml for that matter) only amibiguous ones (like `null` and `map x to x`) with types that are determined by context.

Problem 2. [30 points]

(*i*) [15 points] Is the following Typed Jam program typable? Justify your answer either by giving a proof tree (constructed using the inference rules given at the beginning of the exam) or by showing a conflict in the type constraints generated by matching the inference rules against the program text.

```
let foldr := map f,e,l to
               if null?(l) then e
               else f(first(l), foldr(f, e, rest(l)));
in foldr(cons, null, cons(foldr(map x,y to x+y, 0, cons(1,null)), null))
```

> No. It is not typable. The function bound to `foldr` has type $(\alpha \times \beta \rightarrow \beta) \times \beta \times \alpha - list \rightarrow \beta$ but the type variables α and β are not generalized in Typed Jam because Typed Jam does not support parametric polymorphism (and the type schemes [polymorphic types] required to type polymorphic expressions). In the body of the `let`, `foldr` is used with two different typings: `(int`$\times$`int`$\rightarrow$`int)`$\times$`int`$\times$`int-list` $\rightarrow$ `int` in the inner application and `(int`$\times$`int-list`$\rightarrow$`int-list)`$\times$`int-list`$\times$`int-list` $\rightarrow$ `int-list` in the outer one.

In untyped Jam, the preceding program evaluates to the list `(1)` .

(ii) [15 points] Is the same program

```
let foldr := map f,e,l to
               if null?(l) then e
               else f(first(l), foldr(f, e, rest(l)));
in foldr(cons, null, cons(foldr(map x,y to x+y, 0, cons(1,null)), null))
```

typable in Polymorphic Jam? Justify your answer in same way as in part (i).

Yes. The detailed proof derivation is elided, but the subproof generating a type for the right-hand-side of the `foldr` binding generates the type $(\alpha \times \beta \to \beta) \times \beta \times \alpha - list \to \beta$ where α and β are fresh type variables (not in Γ for the typing of the entire program). In the subproof assigning a type to the body of the `let`, `foldr` has the polymorphic type (a type scheme) $\forall \alpha, \beta[(\alpha \times \beta \to \beta) \times \beta \times \alpha - list \to \beta]$ which enables `foldr` to have the two distinct typings described in the solution to part (i).

Problem 3. [25 points]

Convert the following untyped Jam program to CPS. Use the identity function as your top level continuation and do not CPS either nested lets or applications of primitive operations (primitive functions or operators). Note that `let` is recursive.

```
let foldr := map f,e,l to
                if null?(l) then e
                else f(first(l), foldr(f, e, rest(l)));
in foldr(map x,y to x+y, 0, cons(1,null))
```

Your CPS translation simply has to put all calls on program defined functions in tail position.

```
let foldrK := map fK,m,l,k to
                if null?(l) then k(e)
                else foldrK(f, e, rest(l), map v to fK(first(l), v, k));
in foldrK(map x,y,k to k(x+y), 0, cons(1,null), map v to v)
```

Problem 4. [10 points]
Convert the program

```
let foldr := map f,e,l to
               if null?(l) then e
               else f(first(l), foldr(f, e, rest(l)));
in foldr(map x,y to x+y, 0, cons(1,null))
```

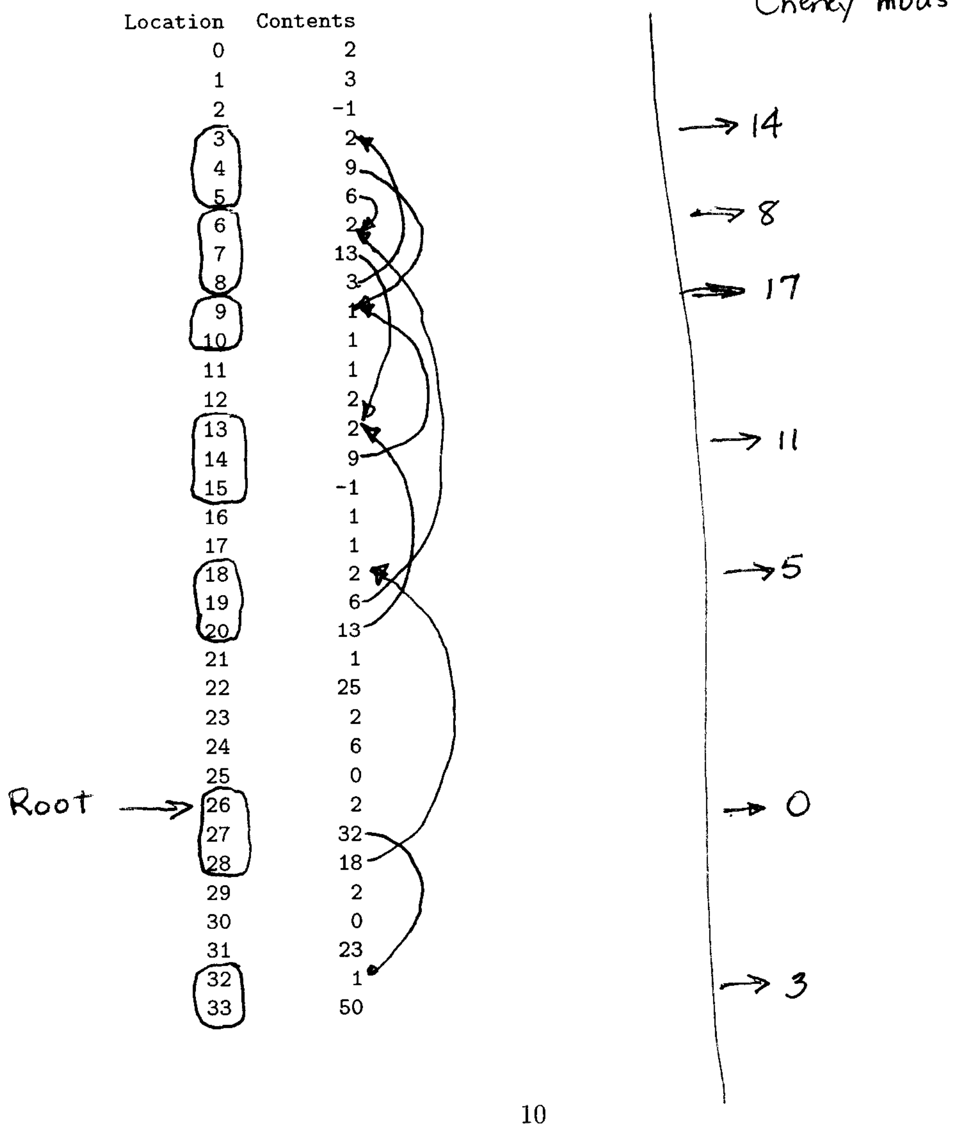
from the preceding problem (before CPS conversion!) to use static distance coordinates instead of symbolic variable references. Recall that static distance coordinates are pairs of natural numbers.

```
let [*1*] map [*3*] to
            if null?(0:2) then 0:1
            else (0:0)(first(0:2), (1:0)(0:0, 0:1, rest(0:2)));
in (0:0)(map [*2*] to (0:0)+(0:1), 0, cons(1,null))
```

Problem 5. [10 points]

Assume that the heap array (of 32-bit machine words) shown below contains two kinds of records: `INT` records and `CONS` records. An `INT` record represents a 32 bit integer *i*; it consists of a tag word containing the value 1 followed by a word containing the 32-bit integer *i*. A `CONS` record represents a pair of data values which are either references to `INT` records, references to `CONS` records, or the null reference. It consists of a tag word containing the value 2 followed by two words containing references. References to heap objects are simply their locations (offsets in words from the base) in the heap. The null reference is represented by the value -1.

Given a root set consisting of location 26, circle the locations of all the live objects in the heap consisting of the following 34 words of memory.



	Location	Contents	Cheney mods
	0	2	
	1	3	
	2	-1	
	3	2	→ 14
	4	9	
	5	6	
	6	2	→ 8
	7	13	
	8	3	
	9	1	→ 17
	10	1	
	11	1	
	12	2	
	13	2	→ 11
	14	9	
	15	-1	
	16	1	
	17	1	
	18	2	→ 5
	19	6	
	20	13	
	21	1	
	22	25	
	23	2	
	24	6	
	25	0	
Root →	26	2	→ 0
	27	32	
	28	18	
	29	2	
	30	0	
	31	23	
	32	1	→ 3
	33	50	

Problem 6. [10 points]

The small heap in the preceding problem is full; there is no unallocated space at the end of the heap array. Using Cheney collection, copy the live objects from this heap into a new heap and set the variable `free` to point to the first free location in the new heap.

Location	Contents
0	2
1	~~32~~ 3
2	~~18~~ 5
3	1
4	50
5	2
6	~~6~~ 8
7	~~13~~ 11
8	2
9	~~13~~ 11
10	~~3~~ 14
11	2
12	~~9~~ 17
13	-1
14	2
15	~~9~~ 17
16	~~6~~ 8
17	1
18	1
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	

`free =`

Problem 7. [10 points]

In problem 5, you marked the live nodes in the heap. Assume that you have recorded this information in a separate bit-map table (only 34 bits long for this tiny heap). Perform the the "sweep" step for a "mark-and-sweep" collector that does not move data and links the free blocks in a free-list where the first word in each free block is the size of the block in words minus 1 and the second word is the address of the next free block. The mininum size for a block in the free list is two words. Coalesce adjacent free blocks and use the dummy pointer value -1 to terminate the list. Set the variable `free` to point the first node in the free list. You do not have to show the bit-map table since it simply records the information given in your answer to problem 5.

Note: if a node is freed and it is isolated (no free node is adjacent), then the header already contains the correct value for the free-list!

free →

Location	Old Contents	New Contents (if changed)
0	2	
1	~~3~~	11
2	-1	
3	2	
4	9	
5	6	
6	2	
7	13	
8	3	
9	1	
10	1	
11	1	
12	~~2~~	16
13	2	
14	9	
15	-1	
16	1	
17	~~1~~	21
18	2	
19	6	
20	13	
21	~~1~~	4
22	~~28~~	29
23	2	
24	6	
25	0	
26	2	
27	32	
28	18	
29	2	
30	~~0~~	-1
31	23	
32	1	
33	50	

Note. free = 0