CS 421 Spring 2012 Practice Final Exam, Part 2

May 5, 2012

1. (*Compilation of MiniJava*)

   Below, you are to give the rules for a `try/catch/finally` block, which does exception handling in Java. Keep in mind the semantics of this statement:

   - Unlike regular Java, only one `catch` block is allowed here. Ignore code for constructing exception objects and focus on control flow.
   - Code within a `try` block executes normally unless an exception is thrown. If this happens, control jumps to the `catch` block (if present), `finally` block (if present), or to the end of the `try` block (if neither are present).
   - Code inside the `catch` block is not executed unless an exception is thrown. Code inside the `finally` block is always executed regardless if an exception is thrown.
   - If an exception is thrown inside a `finally` block, it is not handled - it is thrown back up the control stack.
   - If an exception is thrown inside a `catch` block, control goes to the `finally` block (if present), then thrown, or simply thrown if the `finally` block is not present.

   Recall that compilation rules for statements take the following form:

   $$S,\ m \rightsquigarrow il,\ m'$$

   To implement support for exceptions, we must change the format of statement rules to take an additional location, *ex*, which indicates where to jump to if there is an exception thrown within the statement. (*ex* is a location *in the instruction list*, not a location in the stack.) Use this new format to implement the following questions below:

   $$S,\ ex,\ m \rightsquigarrow il,\ m'$$

   To get you started, here is the compilation rule for the `try` block with no accompanying `catch` or `finally` blocks:

   `try` $S,\ ex,\ m \rightsquigarrow il,\ m'$
   $\quad S,\ m',\ m \rightsquigarrow il,\ m'$

   (a) Give the compilation rule for the `try/finally` block with no accompanying `catch` block.

   `try` S `finally` $S_f,\ ex,\ m \rightsquigarrow$

(b) Give the compilation rule for the `try`/`catch` block with no accompanying `finally` block.

`try` S `catch` $S_{ex},\ ex,\ m \rightsquigarrow$

(c) Give the compilation rule for the full `try`/`catch`/`finally` block. When executing in the `finally` block, you have to remember whether an exception was thrown, because in that case, the `finally` block has to rethrow it (i.e. jump to x); assume that a special stack location th exists for this purpose, that it is initialized to 0 just before the block starts its execution, and that it is not used again after this block's execution is finished.)

`try` S `catch` S_{ex} `finally` $S_f,\ ex,\ m \rightsquigarrow$

2. (*Object-oriented programming, inheritance, and v-tables. This question is from midterm 2.*)

For each of the following Java class definitions, fill in its "v-table" (virtual function table). Each entry should have the form "<function name> in <class name>", meaning this table entry points to the definition of <function name> given in <class name>. The functions in each table should appear in the correct order, as they would in a v-table for Java or C++. We have given the first one.

```
class B {
   void f() {}
   void g() {}
}
```

f in B
g in B

```
class C1 extends B {
   void h() {}
}
```


```
class C2 extends B {
   void g() {}
}
```


```
class D extends C1 {
   void i() {}
   void g() {}
}
```


3. (*Higher-order functions: know how to use existing higher-order functions* `map` *and* `fold_right`*, and how to define your own higher-order functions.*

   The function `map2` is a version of map that iterates over two lists (of the same length) simultaneously. Here is its definition:

```
(* This definition assumes |lis1| = |lis2| *)
let rec map2 f lis1 lis2 -> match (lis1,lis2) with
    ([],[])   -> []
  | (h::t, h'::t') -> f h h' :: map2 f t t'
```

   (a) Give the (polymorphic) type of `map2`

   (b) Use `map2` to define `zip`, which combines two lists into a list of pairs, e.g.

   `zip [1;2;4] [true;false;true] = [(1,true);(2,false);(4,true)]`

   (c) Use `map2` to define `sumelts`, which adds elements of two lists pairwise, e.g.

   `sumelts [1;2;4] [5;7;3] = [6;9;7]`

4. `fold_right` is a higher-order function that encapsulates the notion of "recursion on the tail." Its first argument is a function that combines the head of a list with the result of the recursive call to give the result for this list. Its second argument is the list itself and the third is the value to return for the empty list. Here is its definition:

```
let rec fold_right f lis z =
      if lis=[] then z else f (hd lis) (fold_right f (tl lis) z)
```

   (a) Give the (polymorphic) type of `fold_right`:

   (b) Use `fold_right` to define `unzip`: $(\alpha * \beta)$ list $\rightarrow$ (α list * β list), which is the inverse of `zip`, e.g.

   `unzip [(1,true);(2,false);(4,true)] = ([1;2;4], [true;false;true])`

(c) Use `fold_right` to define `compress`, which takes two lists, one with booleans and the other with arbitrary values, and the returns the list consisting of all the elements from the second list in positions where the first list has `true`, e.g.

```
compress [true;false;true] [1;2;4] = [1; 4]
```

We actually apply `fold_right` only after zipping the the two arguments. Fill in the blank:

```
let compress l1 l2 = fold_right (                                          )

                                 (zip l1 l2) []
```

5. In question 2 from part 1 of the practice problems, you defined functions on a type of data called a "varmap," which mapped identifiers to renamed identifiers. It had three functions (they have slightly different definitions here):

   - `emptyvarmap`: string → int returns zero for all arguments
   - `update x m` returns a new varmap in which `x`'s value is one greater than in `m`
   - `fetch x m` returns a string consisting of `x` concatenated to its value in `m`.

   For example:

```
# let m = update "a" (update "b" (update "a" emptyvarmap));;
# fetch "a" m;;
"a2"
# fetch "b" m;;
"b1"
# fetch "c" m;;
"c0"
```

   In part 1, we used a standard representation of this map as a list of string * int pairs. Here, you are to use a functional representation. We have given you the definition of `emptyvarmap`:

```
type varmap = string -> int

let emptyvarmap = fun s -> 0

let update (s:string) (m:varmap) : varmap =


let fetch (s:string) (m:varmap) : string =
```

6. (*Type systems: MiniJava; monomorphic and polymorphic type systems for OCaml.*)

Here is the explicitly-typed, polymorphic type system from lecture 25:

(Const) $\Gamma \vdash \text{Int i} : \text{int}$

(Var) $\Gamma \vdash a : \Gamma(a)$
($\Gamma(a)$ a type)

(Fun) $\Gamma \vdash \text{fun } a{:}\tau \texttt{ -> } e : \tau \rightarrow \tau'$
$\Gamma[a{:}\tau] \vdash e : \tau'$

(δ) $\Gamma \vdash e \oplus e' : \tau''$
$\Gamma \vdash e : \tau$
$\Gamma \vdash e' : \tau'$

(App) $\Gamma \vdash e\ e' : \tau'$
$\Gamma \vdash e : \tau \rightarrow \tau'$
$\Gamma \vdash e' : \tau$

(True) $\Gamma \vdash \text{true} : \text{bool}$

(False) $\Gamma \vdash \text{false} : \text{bool}$

(PolyVar) $\Gamma \vdash a[\tau] : \tau$
where $\tau \leq \Gamma(a)$
($\Gamma(a)$ a type scheme)

(Let) $\Gamma \vdash \text{let } a{:}\tau \texttt{ = } e \texttt{ in } e' : \tau'$
$\Gamma \vdash e : \tau$
$\Gamma[a{:}\text{GEN}_\Gamma(\tau)] \vdash e' : \tau'$

(a) Write the full proof of this type judgment:

$\emptyset \vdash$ `let g:((int->alpha)->alpha) = fun f:(int->alpha) -> f 4`
`in g[(int->int)->int] (fun x:int -> x+1)` : int

(b) The following judgment was given in lecture 25 to illustrate a flaw in the above type system, when imperative operations are included:

$\emptyset \vdash$ `let i = fun x -> x`
`in let fp = ref i in (fp := not;  (!fp) 5)` : int

This judgment is provable in our type system (even though it shouldn't be, because it has a run-time type error). In this problem, you are to give the *type-annotated* version of this expression, and then prove the judgment using the polymorphic type system).

Recall the polymorphic types of the imperative operations:

`ref` : $\forall\tau.\tau \rightarrow \tau$ `ref`
`!` : $\forall\tau.\tau$ `ref` $\rightarrow \tau$
`:=` : $\forall\tau.\tau$ `ref` $* \ \tau \rightarrow$ `unit`
`;` : $\forall\tau.$ `unit` $* \ \tau \rightarrow \tau$

We have started the explicit type annotation; finish it and fill in the proof (note that, technically, we should also be adding type instances to the uses of the operators `:=`, `!`, etc., but we are allowing you to omit those):

$\emptyset \vdash$ `let i:alpha->alpha = fun x:alpha -> x`
`in let fp:                    = ref i[            ]`
`in (fp[            ] := not;  (!fp[            ]) 5)` : int

7. (*Loop invariants*)

   In the following questions, we provide you with the pre- and post-conditions of a loop, and you are to provide (a) a loop invariant, and (b) a termination function. You do not need to provide that the invariant is an invariant, or that the termination function has the properties required of a termination function.

   (a) $x > 0 \wedge y > 0 \wedge d = 0 \wedge r = x$ {
   ```
   while (r >= y) { r = r - y; d = d + 1; }
   ```
   } $d = x/y \wedge r = x \bmod y$

   Invariant:

   Termination function:

   (b) Here, `a` is an array of length $n > 0$ (indexed from zero):

   $s = 0 \wedge i = 0 \wedge n > 0$ {
   ```
   while (i != n) { s = s + a[i]; i = i + 1; }
   ```
   } $s = \Sigma_{j=0}^{n-1} a[j]$

   Invariant:

   Termination function:

   (c) Again, `a` is an array of length $n > 0$. The function `min(a,i,j)` returns the index of the minimum value in `a` between indices `i` and `j`, inclusive.

   $i = 0 \wedge n > 0$ {
   ```
   while (i != n-1) { k = min(a,i,n-1); t=a[i]; a[i]=a[k]; a[k]=t; i=i+1; }
   ```
   } $\forall 0 \leq m < n - 1.\, a[m] \leq a[m+1]$

   Invariant:

   Termination function: