Programming Languages (COP 4020/6021) [Spring 2017]

Assignment VI

Objectives

1. To demonstrate an understanding of language constructs related to algebraic data types: product, sum, and iso-recursive types.
2. To demonstrate an understanding of evaluation contexts.
3. To implement a type checker and interpreter for diMLazy-Au.

Due Date: Monday, April 24, 2017 (at 5pm).

Machine Details: Complete this assignment by yourself on the following CSEE network computers: c4lab01, c4lab02, ..., c4lab20. These machines are physically located in ENB 220. Do not use any server machines like grad, babbage, sunblast, etc. You can connect to the C4 machines from home using SSH. (Example: Host name: *c4lab01.csee.usf.edu* Login ID and Password: <your NetID username and password>) You are responsible for ensuring that your programs compile and execute properly on these machines.

Assignment Description

Let's consider an extended version of diMLazy. The extended language is called **diMLazy-Au** (pronounced dim-el-azy-gold). The *Au* indicates that the language has *a*lgebraic and *u*nit types. The algebraic types are binary product and sum, and iso-recursive, types.

Begin by downloading and studying http://www.cse.usf.edu/~ligatti/pl-17/as6/diMLazy-Au.sml, which defines the diMLazy-Au datatypes and provides some example diMLazy-Au expressions.

Special for undergraduates: Undergraduates can obtain full credit for this assignment without handling recursive types. Undergraduates who wish to do the basic version of this assignment, without extra credit, should delete all parts of the downloaded diMLazy-Au file that deal with recursive types (specifically, lines 6, 23, 33, and 78-154) and add any missing semicolons. Correctly doing the full assignment, with recursive types, is worth +40% extra credit for undergraduates. Please note that if *any* of your functions handle recursive types, your submission will be graded at the graduate level (meaning all functions are required to handle recursive types).

Note about uniqueness of variable names: In this assignment, variable names are *not* assumed to be unique. Your functions should work correctly even when an expression redeclares an already declared variable. For example, a function is ill-typed if its argument and parameter names are the same, but an inner function may have the same name as a parameter of an outer function. It's also important for graduate students, and undergraduates doing the extra credit, to note that this non-uniqueness extends to type variables—type variables may not be uniquely named.

Note about variable capture: Throughout this assignment, assume all variable names have been chose to avoid capture. Your code should never consider captures nor raise a Capture exception.

Notes about evaluation in diMLazy-Au: The datatype `ctxt` in *diMLazy-AU.sml* shows how the right-to-left evaluation order affects diMLazy-Au expressions. Now let's consider which expressions are affected by diMLazy-Au's CBN evaluation strategy. As discussed in class, CBN affects function applications. It also affects `FstExpr`s, `SndExpr`s, `CaseExpr`s, and `UnrollExpr`s. For example, the expression (e_1,e_2).fst can immediately step to e_1, even if e_1 isn't a value.

Your task is to create a new file called *as6.sml* and implement the following functions in it.

(1) `tc : expr -> typ option`
This function takes a diMLazy-Au expression *e* and returns NONE iff *e* is an ill-typed program and SOME *t* iff *e* is a well-typed program having type *t*.

(2) `isVal : expr -> bool`
This function takes a diMLazy-Au expression e and returns true iff e is a value.

(3) `fill : ctxt -> expr -> expr`
This function takes a valid diMLazy-Au evaluation context *E* and expression *e* and returns *E[e]*.

(4) `decompose : expr -> ctxt * expr`
This function takes an expression *e* and returns an *(E,e')* such that *e=E[e']* and *e'* can take a beta step. If no such *(E,e')* exists, then this function raises the *Stuck* exception.

(5) `beta : expr -> expr`
This function returns the result of β-stepping its argument; if no β-step is possible, *Stuck* is raised. This beta function should have a helper function called `sub`, which implements [e/x]e'.

(6) `smallStep : expr -> expr`
This function returns the result of stepping its argument; if no step is possible, *Stuck* is raised. This small-step operation must be defined in terms of evaluation contexts (as discussed in class).

(7) `bigStep : expr -> expr`
This function returns the value resulting from fully evaluating its argument expression *e*. If *e* gets stuck, this function raises *Stuck*, and if *e* diverges, so does this function. This big-step operation must be defined in terms of small-step operations.

Every one of the last six functions (numbered (2)-(7) above) must be legitimately invoked somewhere in `smallStep` or `bigStep`.

For full credit, your implementation must also:

- be commented and formatted appropriately (as on previous assignments).
- use ML features like pattern matching when appropriate.
- not define any extra top-level values.
- compile and run on the C4 machines with no errors or warnings (except for Stuck exceptions raised at appropriate times).
- not use any library functions.
- not use any ML features that cause side effects to occur (e.g., I/O or references/pointers).
- not be significantly more complicated than necessary.
- be reasonably efficient.

The submission process is the same as for other programming assignments, except here you'll submit *as6.sml* in Canvas. Please remember to include the pledge as an initial comment; not doing so will lower your grade 50%. As usual, you may submit this assignment up to 48 hours late with a 15% penalty.

Hints: My *as6.sml* is 192 lines of code (not counting comments and whitespace) and took me about 8 hours to implement and test (with most of the time going to recursive types).

Sample Executions (but we'll test your code on other examples as well)

```
> rlwrap sml
Standard ML of New Jersey v110.74 [built: Thu Aug 16 11:25:45 2012]
- use "diMLazy-Au.sml";
...
- use "as6.sml";
[opening as6.sml]
val tc = fn : expr -> typ option
val isVal = fn : expr -> bool
val fill = fn : ctxt -> expr -> expr
val decompose = fn : expr -> ctxt * expr
val beta = fn : expr -> expr
val smallStep = fn : expr -> expr
val bigStep = fn : expr -> expr
val it = () : unit
- tc e1;
val it = SOME (Arrow (Sum (Unit,Prod (Int,Int)),Sum (Prod (Int,Int),Unit)))
  : typ option
- tc e2;
val it = SOME (Sum (Unit,Prod (Int,Int))) : typ option
- tc e3;
val it = SOME Int : typ option
- decompose e2;
val it =
  (SumCtxt (Right,PairCtxt2 (Hole,IntExpr 6),Sum (Unit,Prod (Int,Int))),
   IfExpr (FalseExpr,IntExpr 4,IntExpr 5)) : ctxt * expr
- e2 = (fill (#1 it) (#2 it));
val it = true : bool
- smallStep e2;
val it =
  SumExpr (Right,PairExpr (IntExpr 5,IntExpr 6),Sum (Unit,Prod (Int,Int)))
  : expr
- isVal it;
val it = true : bool
- bigStep e3;
val it = IntExpr 5 : expr
- smallStep it;

uncaught exception Stuck
  raised at: as6.sml:176.25-176.30
- tc (FunExpr ("f","x",Prod(Int,Bool),Int,IntExpr(4)));
val it = SOME (Arrow (Prod (Int,Bool),Int)) : typ option


- (* Now for some tests related to recursive types *)
- tc (FunExpr ("f","x",Prod(Int,Var("t")),Int,IntExpr(4)));
val it = NONE : typ option
- tc e4;
val it =
  SOME
    (Arrow
       (Rec ("t",Sum (Unit,Prod (Int,Var "t"))),
        Rec ("t",Sum (Unit,Prod (Int,Var "t"))))) : typ option
- tc e6;
val it = SOME (Rec ("t",Sum (Unit,Prod (Int,Var "t")))) : typ option
```

```
- bigStep e5;  (* this is the list 2::3::4::nil *)
val it =
  RollExpr
    (SumExpr
       (Right,
        PairExpr
          (IntExpr 2,
           RollExpr
             (SumExpr
                (Right,
                 PairExpr
                   (IntExpr 3,
                    RollExpr
                      (SumExpr
                         (Right,
                          PairExpr
                            (IntExpr 4,
                             RollExpr
                               (SumExpr
                                  (Left,UnitExpr,
                                   Sum
                                     (Unit,
                                      Prod
                                        (Int,
                                         Rec
                                           ("t",Sum (Unit,Prod (Int,Var "t")))))))),
                          Sum
                            (Unit,
                             Prod
                               (Int,Rec ("t",Sum (Unit,Prod (Int,Var "t")))))))),
                 Sum (Unit,Prod (Int,Rec ("t",Sum (Unit,Prod (Int,Var "t")))))))),
        Sum (Unit,Prod (Int,Rec ("t",Sum (Unit,Prod (Int,Var "t"))))))) : expr
- bigStep e6;  (* this should reverse e5, to produce 4::3::2::nil *)
val it =
  RollExpr
    (SumExpr
       (Right,
        PairExpr
          (IntExpr 4,
           RollExpr
             (SumExpr
                (Right,
                 PairExpr
                   (IntExpr 3,
                    RollExpr
                      (SumExpr
                         (Right,
                          PairExpr
                            (IntExpr 2,
                             RollExpr
                               (SumExpr
                                  (Left,UnitExpr,
                                   Sum
                                     (Unit,
                                      Prod
                                        (Int,
                                         Rec
                                           ("t",Sum (Unit,Prod (Int,Var "t")))))))),
                          Sum
                            (Unit,
                             Prod
                               (Int,Rec ("t",Sum (Unit,Prod (Int,Var "t")))))))),
                 Sum (Unit,Prod (Int,Rec ("t",Sum (Unit,Prod (Int,Var "t")))))))),
        Sum (Unit,Prod (Int,Rec ("t",Sum (Unit,Prod (Int,Var "t"))))))) : expr
```