Implementing Classes, Arrays, and Assignments

Logistics

–PA4 peer reviews are due Saturday

–HW9 is due Monday

–PA5 is due December 5th

–Will talk about monad implementation at some point, until then check out paper "Imperative functional programming" by Simon L. Peyton Jones and Philip Wadler if you are curious.

Implementing Classes, Arrays, and Assignments

(1) Memory model for classes and arrays

(2) Type Checking

(3) Code Generation

PA5 Overview

Goals

- Expansion of lexer, parser, AST, and symbol table for objects, assignment statements, and arrays
- Type checking for objects, assignment statements, and arrays
- Code generation for objects, assignment statements, and arrays

New pieces of grammar

- Variable declarations
- Assignment statements
- Object creation
- Member variables
- Array creation and usage

Objects (aka records, structs, ...)

An object is a collection of data, usually related in some way

- Each piece of data might have a name (or field name)
- Haskell data types can use position or names

```
data Point = Pt Float Float
  -- OR
data Point = Pt {pointx, pointy :: Float}
pointx :: Point -> Float
pointy :: Point -> Float
```

Object memory model

- Object instances are created with heap allocations.
- Each object instance places fields in same location.

Exercise: draw a memory map (RTS and heap)

```
class PA5obj {
     public static void main(String[] whatever) {
         new C().setP((byte)3,(byte)7,Meggy.Color.BLUE);  } }
class C {
     Ind oy;
     public void setP(byte x, byte y, Meggy.Color c) {
        Ind ox; ox = new Ind(); ox.put(x);
        oy = new Ind(); oy.put(y); /* Here 3 */  }       }
class Ind{
     byte _i;
     public void put(byte i){ _i = i;    /* Here 1,2 */   }
     public byte get(){ return _i; }       }
```

1: just after **ox.put()** has executed (but not returned)

2: just after **oy.put()** has executed (but not returned)

3: just after **oy.put()** has returned

Arrays

An array is a collection of items of the same type

- so that the address of an element can be computed from the start address and the index (efficiency)
- index: int (or int derivative type like unsigned or byte)

Once an array is allocated, the sizes of its dimensions do not change (as opposed to ArrayLists, Lists, …)

Java arrays are one dimensional

- higher dimensional arrays are arrays of arrays

  these are sometimes called "ragged" arrays, as the lengths sub arrays can differ
- as opposed to rectangular arrays in Fortran

Array representations

1. store length with array elements

e.g. at the front of the array

- this is nice for Java arrays

the array is now represented by its start address

- the address of the length field

when allocating and indexing in such arrays this length field must be taken into account (added / skipped over)

2. Have a separate array descriptor with

index ranges for all dimensions

widths in bytes in each dimension

(some representation of) start address

PA6Rainbow.java

```
class PA6rainbow { public static void main(String[] whatever){{
        // display a rainbow on row 5
        new Rainbow().run((byte)5); }}}
class Rainbow {
   Meggy.Color [] p;
   public void run(byte row) {
      p = new Meggy.Color [8];
      p[0] = Meggy.Color.RED;
      p[1] = Meggy.Color.ORANGE;
      p[2] = Meggy.Color.YELLOW;
      p[3] = Meggy.Color.GREEN;
      p[4] = Meggy.Color.BLUE;
      p[5] = Meggy.Color.VIOLET;
      p[6] = Meggy.Color.WHITE;
      p[7] = Meggy.Color.DARK;
      Meggy.setPixel((byte)2, (byte)3, p[0]);
      Meggy.setPixel((byte)2, (byte)4, p[4]);
      this.displayRow(row, p);
   }
   public void displayRow(byte row, Meggy.Color [] a) {
      int i;  i=0;
      while (i<8) {
         Meggy.setPixel((byte)i, row, a[i]);       i = i+1;
      } }    }
```

Implementing type checking for PA5 MeggyJava

Visitor over AST will check for type errors at each AST node

Syntax *AST node*

`id = Exp ;` `AssignStatement(id, Exp)`

[LINENUM,POSNUM] Undeclared variable VARNAME

[LINENUM,POSNUM] Invalid expression type assigned to variable VARNAME

`public Type name(...) {...return Exp; }` `MethodDecl(name, Stms, Exp)`

[LINENUM,POSNUM] Invalid type returned from method METHODNAME

Exp . name (Args) CallExp(name, Args)

[LINENUM,POSNUM] Receiver of method call must be a class type

[LINENUM,POSNUM] Method METHODNAME does not exist

[LINENUM,POSNUM] Method METHODNAME requires exactly NUM arguments

[LINENUM,POSNUM] Invalid argument type for method METHODNAME

Error message for symbols redeclared within same scope

```
Class ID …                          ClassDecl
public Type ID …                    MethodDecl
Type ID;                            VarDecl
```

[LINENUM,POSNUM] Redefined symbol VARNAME

Code Gen for Classes and Local variables

Method activation records on run-time stack

- Parameters will still have locations in the activation record.
- Local variables will also have locations in the activation record.

Member variables will be stored in object instances

- The new expression should cause a call to malloc.
- Member variables will have offsets within an object instance.
- The "this" variable will contain a pointer to the object instance.

BuildSymTable for varDecl

VarDecl

create VarSTE

if it is a member variable

make the base "Z" (r31:r30)

make the offset the current class offset (ClassSTE will need this)

increment the class offset/size with the size of the variable

else if it is a local

make the base "Y" (r29:r28)

make the offset the current method offset (MethodSTE)

increment the method offset/size with the size of the variable

Code Generation for method call and this

CallExp

1) Use checkTypes to determine class type or receiver.
2) Look up the ClassSTE from the receiver type.
3) Then lookup the MethodSTE from the ClassSTE scope.
2) Generate code that pops parameters off the stack and
   into the appropriate registers from right to left.
   Receiver reference is the first parameter (this), last pushed.
3) Generate code that calls the mangled method name.
4) Generate code that pushes the return value back on the stack.

ThisExp

1) push the value of the "this" parameter onto the run-time stack
   load “this” into r31:30 and then push it

Code Generation for IdExp and assignStmt

IdExp

1) Lookup id in symbol table to get VarSTE
2) If the VarSTE is a member variable
   2a) Look up VarSTE for "this" and generate code
      that loads the value of "this" into registers r31:r30.
3) load variable into a register(s) using the base+offset from VarSTE.
4) Push the variable value on the stack.

AssignStatement

1) Lookup id in symbol table to get VarSTE
2) If the VarSTE is a member variable
   2a) Look up VarSTE for "this" and generate code
      that loads the value of "this" into registers r31:r30.
3) store value of expression on top of run-time stack into base+offset from VarSTE