Project 2 Hints

Jingke Li

Portland State University

Basic Information

- *The Infrastructure* —

  Same as Project 1's, but with two more packages:

  - `ast` --- the source-language's AST nodes
  - `astpsr` --- the AST parser program
  - `symbol` --- the symbol table package
  - `checker` --- the typechecking package
  - `ir` --- the IR tree nodes
  - `irgen` --- the irgen code
  - `tst` --- a set of test programs

- *New Issues* —
  - new objects
  - method declarations
  - method invocations
  - object fields
  - "this" node
  - variable references

New Objects

A `NewObj` node should be translated into a call to `"malloc"` with a single argument representing the size of the object, followed by a sequence of statements for initializing its instance variables.

Note that the initialization expressions saved in the the symbol table need to be translated into `IR` expressions before they can be used to initialize the instance variables. This translation should be conducted in the proper scope environment — the `IrgenVisitor` variable `currClass` needs to point to the object's class.

```
public EXP visit(NewObj n) throws Exception {
  currClass = <get the object's ClassRec from symbol table>;
  ExpList inits = currClass.initExps();
  for (int i=0; i<currClass.vcnt(); i++) {
    <process an init expr, and construct a MOVE node to
     copy it to the corresponding instance variable's slot>
  }
  ...
  return new ESEQ(stmts, obj);
}
```

Method Declarations

- Create an unique label for the method by concatenating class name and method name together, with an underscore (_) in between.
  - Look up the corresponding `MethodRec` in the symbol table. To do this, you need to know what class this method decl belongs. (Recall the use of variable `currClass` in the typechecking project).

    ```
    MethodRec currMethod = currClass.getMethod(n.mid);
    ```

  - Call the "`uniqueMethodName()`" method of the symbol table.

    ```
    String label =
      symTable.uniqueMethodName(currClass, currMethod.id());
    ```

  Note that for the `main` method, this step is skipped.

Method Declarations (cont.)

- Compute the values of `varCnt` and `argCnt`.

  `varCnt` is the local variable count, and `argCnt` is the maximum number of arguments of a single `Call/CallStmt` node (excluding calls to system routines) in current method's body. Both values are needed in the later Codegen module for calculating frame size.

  - You can compute `varCnt` by calling the "`localCnt()`" method of the corresponding `MethodRec`.
  - The value of `argCnt` has to come from recursive visit to the method's body. You need to declare a class variable, say `maxArgCnt`, at the top level, and if necessary, update the value every time a `Call` or `CallStmt` node is encountered:

    ```
    if (maxArgCnt <= argCnt)
      maxArgCnt = argCnt + 1;
    ```

    In the above code, argCnt is the Call node's arg count, and the extra 1 is for passing the 'access link'.

Method Invocations

- Create the same unique label for the method.

  Again, you should look up the symbol table for the `MethodRec` and use the "`uniqueMethodName()`" method. However, you first need to figure out the class that the method belongs to (which could be a different class than `currClass`. The proper steps are:

  1. Perform typechecking on the `obj` component of the `Call/CallStmt` node, which should return an `ObjType` representing the `obj`'s class.

     An Important Detail: The `TypeVisitor` program has its own copy of `currClass` and `currMethod`. These two variables need to be set to the current environment before typechecking can work. Do the following:

     ```
     typeChecker.setClass(currClass);
     typeChecker.setMethod(currMethod);
     ```

  2. Look up the `MethodRec` with the class info.
  3. Finally, concatenate class name and method name together.

Method Invocations (cont.)

- Add a pointer to the object record as the 0th element of the argument list (*i.e.* set up the access link).

```
EXPlist el = new EXPlist();
el.add(n.obj.accept(this));
```

`CallStmt` is translated similarly, except that it returns a `CALLST` node.

Object Field and "This" Node

- Object field in the form of a `Field` node is translated into a `FIELD` node

  ```
  (Field obj var)  =>  (FIELD <obj's record> <var's idx>)
  ```

  Note that to compute the variable's index, you need to know which class the variable belongs to. Do similar things as in `Call` nodes, i.e. performing typechecking on the `obj` component, and get the class info.

- "This" node should point to the current class object. We assume that it is only used within a method's body. Hence it should be translated into `PARAM(0)`.

Variable References

Standing-along variables, as represented by `Id` nodes, are translated into `FIELD`, `PARAM`, or `VAR` nodes; all with an index.

- A standing-alone class variable is translated into a `FIELD` node referencing from the current object:

  ```
  (FIELD PARAM(0) <var's idx>-1)
  ```

  In this case, it is equivalent to having a `"This."` prefix in front of the variable.

  Note that when an `Id` node appears inside a `Field` node, it is taken care of by the visit routine of the `Field` node. In that case, there is no need to recursively visit the `Id` node.

- A parameter is simply translated into a `PARAM` node.
- A local variable is simply translated into a `VAR` node.

A variable's index value can be looked up from the symbol table with the "`v.idx()`" method.