Programming Assignment 4 - Intermediate Code Generation

Collaborators

- Michael Maitland, mtm68
- Scott Bass, sb2383
- Michael Tobin, mat292

Running the Program

Main class: `mtm68.Main.java`

Run `xic-build` (this requires having Maven installed). Then `xic [options...] <source-files>` becomes available.

Summary

An interesting aspect of this assignment was the ability to divide the overall objective into a few isolated passes over either the AST or the IRCode. This allowed us to each focus on a pass or two as well as thoroughly test each pass alone. This made debugging easier as we could isolate in which stage the problem was occurring. This also allowed us to leverage the Visitor design pattern we had used in the previous assignment as well as the one provided to us in the release code. It did challenge us in that it required us to communicate frequently and clearly to make sure that everyone was operating under the appropriate assumptions. Overall, this helped us easily organize the assignment as well as the tests we created for our code. This gives us high confidence that our implementation is correct.

Specification

The following are choices we made regarding specification:

AST to IR Conversion

- `Expr`s have a field `IRExpr` and `Statements` have a field called `IRStmt`.
- `FunctionDefn`, which is neither an `Expr` nor a `Statment` have a field called `IRFuncDefn`
  - We renamed `IRFuncDecl` to `IRFuncDefn` since `FuncDecl`s do not have a body and the original `IRFuncDecl` contained a body
- `FunctionDecl`, `Interface`, `Use`, and `ErrorStatement` AST nodes are not represented as part of the IR tree
- `SimpleDecl` initializes the declared variable to 0 if no assignment is made

[INSERT]

Design and Implementation

Architecture

The key classes and packages we created or updated for this assignment are the following...

- mtm68.Main.java
  - Our Main functions very similarly to the previous assignment. We added new options to the command line as described in the spec. We also added intermediate code generation to the source file pipeline.
- edu.cornell.cs.cs410.ir.visit.Lowerer
  - This is the IRVisitor who's purpose is to lower the orginal IRCode. At a high level, this visitor tags expressions with their list of sideeffects, turns statements in sequences that include their inner expression's side effects, and flattens SEQs. This allows for there to be no ESEQS (theey become expressions with a list of side effects) as well as a flattened SEQ structure. This Lowerer also takes advantage of commutability in the cases of IRBinOp and IRMove. If the expressions can commute safely, the Lowerer opts for the more efficient code.
- edu.cornell.cs.cs410.ir.visit.ConstantFolder
  - This is the IRVisitor who's purpose is to fold constants in IRCode. The only place where this is possible at the IR level is in the IRBinOp. This visitor, at each IRBinOp, checks to see if both the left and right expressions are IRConsts and, if they are, computes the binop and returns the result in a new IRConst. The only exceptions are when there is a division or modulo by zero in which case the IRBinop is not folded and returned as is.
- mtm68.visit.NodeToIRNodeConverter
  - This is an AST visitor who's purpose is to decorate the AST nodes with their IR transformation. It is also responsible for getting fresh labels, fresh temps, named temps, getting encoded function names.
- edu.cornell.cs.cs410.ir.visit.CFGVisitor
  - This is an IR level visitor whose job is to construct CFGs and perform the re-ordering of statements in `IRSeq`s to ultimately result in canonical form code. The `CFGVisitor` uses the `CFGBuilder` and `CFGTracer` to assist in the process of building graph and re-ordering the statements.
- edu.cornell.cs.cs410.ir.visit.UnusedLabelVisitor
  - This is an IR level visitor that removes unused labels by keeping track of which labels have references to them.
- mtm68.ir.cfg.CFGBuilder
  - This is a helper class that builds a control flow graph as a result of visiting all the statements in an `IRSeq`. It builds the graph out of `CFGNode`s and `CFGEdge`s.
- mtm68.ir.cfg.CFGTracer
  - This is another helper class that takes a CFG and the list of statements

that were used to construct the graph and outputs a new ordering over the statements that leave the code in canonical form. This new ordering removes unnecessary jumps and makes sure all false branches of conditional jumps are fallthroughs.

Code Design

- For this assignment, we essentially were able to use the Visitor pattern to accomplish each subtask we needed to complete. We used our (AST) Visitor to create IRCode. We then used the IRVisitor to lower the IR, constant fold, and handle the basic block reordering. By breaking up the assignment in this way, the work become highly parallelizable as well as relatively straightforward. Each of the translations and transformations themselves were not too complicated. The most interesting case deals with the construction and use of the CFG to reorder basic blocks. This is done by the `CFGVisitor` which relies on two helper classes, `CFGBuilder` and `CFGTracer` to do the re-ordering. After the code is lowered, statements contained within an `IRSeq` call the `CFGBuilder` upon visiting, building the CFG as the visitor traverses. Upon leaving the `IRSeq`, all the nodes are collected from the builder and passed to the `CFGTracer` which then performs the re-ordering, handing back a new set of `IRStmt`s that the `IRSeq` can then use to update. Finally, the `UnusedLabelVisitor` cleans up unused labels.
- The conversion from AST to IR transformations are done in the AST node classes and use the `NodeToIRConverter` when the transformation requires information related to state or wants to do an operation that is reused by other AST nodes. This makes it easy to understand the high level translations while allowing for code reuse.
- One tradeoff we made had to do with the commutability checks at the IR level for IRBinOp and IRMove lowering. We decided, for the sake of simplicity, to consider all IRMem accesses to be alias and did not decide to do further analysis described in lecture. We figure this will likely not have too big of an impact on performance which is why we avoided the hassle.
- The most interesting data structures were used in the CFG construction. In the `CFGBuilder`, the CFG is built using `CFGNode`s. A `CFGNode` reprsents a graph node with a list of incoming `CFGEdge`s and a list of outgoing edges. The graph is double-linked, allowing for easy lookups of incoming and outbound connections. This makes repairing jump nodes in the `CFGTracer` easy.
- For `ExtendedDecl`, we used an IR level function named `_I$allocLayer_piiiiii` to help allocate the arrays properly. The code for this function was handmade and is called upon by the generated IR code for `ExtendedDecl`. The reason we decided to go this route was because since you don't know at compile-time the length of the arrays you're allocating, you have to

use an IR level loop. But, with multi-dimensional arrays, this gets quite complicated since you end up with levels of arrays pointing to other arrays. The function alloc layer is a helper function that wires up all the pointers properly at the IR level. It gets called after the dimensions are all checked to be valid and some offsets and sizes are computed.

Programming

- The greatest challenge during this assignment was keeping each other informed about the individual passes we were working on. It was crucial to make sure we were all on the same page about the assumptions that could be made at each stage of code processing. We each decided to approach our passes in a bottom-up manner. This involved writing transformations for the simpler nodes and expressions and building our way up from there. This allowed us to ensure all of the "base cases" were correct as we built up in complexity.
- The following is the team coding/responsibility breakdown for this assignment...
  - **Tobin:**
    - IRLowerer + tests
    - IRConstantFolder + tests
    - Integration tests
  - **Maitland:**
    - AST to IRCode generation + tests
  - **Bass:**
    - CFGVisitor + tests
    - CFGBuilder + tests
    - Basic Block Reordering + tests
    - UnusedLabelVisitor + tests
- We used our previous code for lexing, parsing, and typechecking. Fortunately, we have been on top of correcting our errors after each assignment so there were very few changes that needed to be made to this code.

Testing

In this assignment, we each found it very useful to follow test-driven development. This allowed us to be confident that our numerous passes were operating correctly for each corresponding node. This not only prevented us from having to rewrite code (since it was often correct on the first go) but also gave us a comprehensive test suite for each pass to rely on when we needed to make changes. This gave us great confidence our implementations were still correct even after changes. We were also able to leverage the IRVisitors in the release code to ensure our lowering and constant folding were successful.

In order to succeed with this plan, we each created a JUnit test suite for

each of the AST/IRCode passes we were responsible for. For each visitor described in the architecture section, you can find a comprehensive test suite in /xic/src/test/java/mtm68/ir. Here, you can find test cases for very simple translations as well as more complicated situations.

A very important part of our testing for this assignment was writing integration tests to ensure the system worked as a whole. This was crucial as we did a lot of the work in parallel (see Work Plan) and had to guarantee our pieces fit well together. This required writing Xi programs and making sure the final IRCode made sense and was correct. To do this, we used the IRCode interpreter provided to us to see that our generated IRCode was computing what it should have been. Using this method revealed a handful of errors in our implementation (such as failure to concat arrays or handling out of bounds errors incorrectly).

It should also be said that we utilized the xth test suite to make sure our output was compliant with the autograder's expectations.

Work plan

For this assignment, we found the work to be highly parallelizable. As this assignment requires creating a few passes over the AST and IRCode, we were able to divide the passes up and work independently on them under the assumptions made at each pass. (For example, after lowering, it can be assumed there are no ESEQs in the IRCode). Maitland handled the generation of IRCode from the AST. Tobin lowered the IRCode and implemented constant folding at the IR level. Bass handled the CFG analysis and reordering of the basic blocks.

In order for our independent work to be successful, we did communicate frequently to ensure that we were all operating on the correct assumptions and that our passes would fit correctly when put together. When it came time to put the pieces together, things came together with just some minor refactoring.

Known Problems

We currently are not aware of any issues with our compiler.

Comments

The release code was very helpful! Thank you!