CS4212 Assignment 3 Report

- Liow Jia Chen
- A0184022R

- 21 November 2020

For a getting started or how to run the compiler refer to the readme instead.

Introduction

This report will detail some of the underlying design decisions when implementing the JLite compiler.

Testcases:

Can be found in `src/test/pa3/pass`

Here is a brief description of the test cases
0. Hello Wolrd

1. Provided Test Case
2. Factorial Function (both recursive and iterative)
3. Function calls (demonstrates passing args via register & stack)
4. Cat Dog OOP (demonstrate class objects and members & string concat)
5. Spilling (force spilling behaviour by having alot of interfering variables)

Phase 1: Lexing and Parsing

For lexing and parsing we used the JFlex and Cup respectively to generate java code via grammar rules.

I had to relax the grammar rules a little to handle string concatenation. For example I allowed `1 + "2"` through. We would flag these errors though type checking in the next phase.

Phase 2: Semantic Analysis

During this phase, we would go over the parse tree and flag any semantic errors. If we observed an errors we would quit and not proceed with compilation.

Check 1: Distinct Name Checking

In this semantic check, we ensured that variable and class have unique names to prevent shadowing.

Method Overloading

Interesting portion here was defining “uniqueness” for overloaded methods. I decided on using a method signature consisting of method name and arg types as per java.
I implemented a mechanism for allowing method overloading by introducing a method signature which includes the method name and the argument type
We allow overloading iff method signature are different.

Method signatures are equal iff method name are the same and arg types are the identical (order matters)

This is derived and consistent from overloading rules in modern languages eg Java.
See implementation of ClassDecl::checkUniqueMethodSignature for implementation details.

Method Call Ambiguity

While coding, I noticed an edge case (see fail test case 6.in (http://6.in)) where a method call could potentially be ambiguous when passing nulls as arguments;
Since it could potentially match many methods. What Java does is try to infer uniqueness by checking how many methods it could possibly be
if its 1 -> carry on and assume that as the method signature
else if its more than one -> throw an error!

```
class Test {
     Int method(Dummy d) {
         return 1;
     }

     Boolean method(Dummy2 d) {
         return true;
     }

     Void test() {
         // throw error, ambiguous method call could be method(Dummy) or method
         // return type is not considered in method resolution since its not pa
         int i = method(null);
     }
 }
```

This is how java works as well!

Check 2: Type Checking

For type checking there was nothing to complex here since we did not have to deal with generics and inheritance!

A nice side effect of this is that for IR and code gen we always have the type information that is useful when creating a new instance of the class via `new` later on!

Phase 3: IR Generation

For IR generation, we followed the IR specification closely.

We generated a 3 address code and SSA form code closer to somewhere in between Jlite and arm. Biggest change would be all control flow statements like ifs and whiles have all be translated to if goto and goto statements via labels.

The trickiest part of this portion was thinking of the right level of abstraction. The hard part was figuring out how to keep track of the temp variables created.

Finally solved this by creating a `TempVariableGenerator` as a global singleton with an atomic index. Also, all statements would generate a list of temp variables and list of ir3 statements which would then get propagated all the way upwards.

Phase 4: Code / Arm Generation

What I loved at this point, was I didn't have to think much about errors and edge cases anymore since all would have been caught by now!

Initial

I first started by constructing the arm code storing all the variables on the stack without and load and store them whenever needed. This provided a good base to implement register allocation.

Lowering IR

At this point, I made the decision to lower IR to make code gen easier. Ideally we would create another IR representation like Gimple but for simplicity i just reworked the current IR.

Changes include

1. remove binary expressions from while and if loops
   `if (n > 1) goto 1;`
   into
   `t1 = n > 1`
   `if (t1) gotot 1;`

2. in expressions
   `x.a = y.b`
   into
   `t1 = x.b`
   `y.a = t1`

3. introducing implicit `this` for class variables
   `x = 123;`
   `println(x);`
   where x is a class variable
   into
   `this.x = 123;`
   `println(this.x);`

This was one of the best decisions I made! Code generation was a lot easier and registers wouldn't clash as much since we can treat each statement as atomic.

Register Allocation

This was perhaps the hardest part of the assignment! But the most fulfilling one as well! Seeing how everything works after graph colouring is simply magical!

Allocation Strategy

a1 and a2 -> scatch registers
a3-4 and v1-5 -> allocated to variables

You can print out the intermediate data structures used such as the interference graph with the `-d` or `--debug` flag when compiling! An example of register allocation for the factorial function can be found in appendix A below.

We implemented the graph colouring algorithm proposed in the lecture. We first build the interference graph via the def use chain. Following this we ran simplify and select algorithm with the max degree cardinality heuristic for spilling.

This produced an allocation of registers to variables with no interference.

For the spilled variables in the function prolog, we reserve space for it on the stack and assign the associated offsets.

Register A3 and A4 were given out last and if there were used, the necessity push / pop operations had to be made before every function call since they can be trashed / clobbered across function calls.

An example allocation for test case 5 where we forced spilling!

```
=== Allocation ===
{_t2=v2, _t1=v1, _t4=v2, _t3=v1, _t6=v2, a=spill, _t5=v1, b=spill, _t8=v2, c=sp
==== End Allocation ===
```

Spilling Heuristic

When choosing which variable to spill first, I used the max degree. This would allow for more freedom after as we relax constraints on more vertices. I also tried the least use heuristic and also a linear combination of these two. In the end i found for my testcases, max degree heuristic performed the best!

Function Calls

Handling function calls were one of the more time consuming parts, getting the offset right took abit of trial and error.

As per arm caller callee convention, if you have too many arguments you have to push them onto the stack instead of registers. As per our implementations is `numberof(args) > 4` we will use the stack.

Fundamentally our call expression and our function prolog / epilogue just had to agree and know when and how to handle each case.

Function Call 1: Register based

In this register based call, arguments are passed via a1-4.

When invoked these arguments are copied over into their allocated register or spill location the stack.

Call 2: Stack based

In this stack based approach we push the arguments in reverse order onto the stack. The pro of this method is that it allows for an arbitrary number of arguments to be pushed onto the stack.

When invoked, the function prolog will copy over all allocated registers into their dedicated registers for arguments that are spilled we just add an entry into the offset table since it already lives on the stack.

After returning from the function, the caller has the responsibility to pop these elements off the stack.

New

For the `new` command, instead of using the prescribed `bl _Znwj(PLT)` I used libc `calloc` instead. This was because I couldn't get gem5 to link the c++ wrapper. `calloc` also initialized the block to zero for me which was nice. eg class int and bool variables initialized to 0 and false respectively.

String Concatenation

For concatinating two string, we leveraged libc functions like `strlen`, `strcpy` and `strcat`. We used `strlen` to compute the length of the new string and then malloc! And then copy the first string over and then cat the second string behind.

```
Given two strings a and b

String res = malloc(strlen(a) + strlen(b));
strcpy(res, a);
strcpy(res, b);
return res
```

Phase 5: Post Optimizations

After generating the arm assembly, we can run the assembly code though some optimization passes with the `-o` flag. These are peep hole optimizations that preserve correctness and behaviour of the code! There are written in regex and run against the code until it converges.

All these local optimizations removes all the unnecessary zig zag patterns in the code.

Optimization 1

Before	After
`str rX, [...]`	`str rX, [...]`
`ldr rX, [...]`	

Trivially the second instruction is redundant!
Note that we can do this wlog for $x,y \in \{r0-8\}$.

Optimization 2

Before	After
`str rX, [...]`	`str rX, [...]`
`ldr rY, [...]`	`mov rY, rX`

Similar to the firs optimization, in the case where registers differ, mov is alot cheaper than a round trip to the cache / memory!
Note that we can do this wlog for $x,y \in \{r0-8\}$.

Conclusion

YAYY I have a working compiler now!

Appendix A: Register allocation for factorial function

```
--- Inteference Graphh (Factorial_0) ---
=== IR WITH INDEX ===
line 0 ->  _t1 = n == 1;
line 1 ->  If (_t1) goto 1;
line 2 ->  _t2 = n - 1;
line 3 ->  _t3 = Factorial_0(this, _t2);
line 4 ->  Return n * _t3;
line 5 ->  goto 2;
line 6 -> Label 1:
line 7 ->  Return 1;
line 8 -> Label 2:
---
0: 1
1: 6, 2
2: 3
3: 4
4: 5
5: 8
6: 7
7: 8
8:
=== TOPO SORTED ===
0:  _t1 = n == 1;
1:  If (_t1) goto 1;
2:  _t2 = n - 1;
3:  _t3 = Factorial_0(this, _t2);
4:  Return n * _t3;
5:  goto 2;
6: Label 1:
7:  Return 1;
8: Label 2:
=== END TOPO SORTED ===
=== FIRST AND LAST USE ===
_t1: first def (0), last use (1)
_t2: first def (2), last use (3)
_t3: first def (3), last use (4)
this: first def (-1), last use (3)
n: first def (-1), last use (4)
=== END FIRST AND LAST USE ===
=== Interference Graph ===
_t1: this, n
_t2: _t3, this, n
_t3: _t2, this, n
this: _t1, _t2, _t3, n
n: _t1, _t2, _t3, this
=== End Interference Graph ===
-----

=== Allocation ===
{this=v4, _t2=v3, _t1=v1, n=v2, _t3=v1}
==== End Allocation ===
```