Implementation Notes and Analysis of Self-Built Compiler for Mini language

CSC 431 - Spring 2020 – Dr Keen

Submitted by

Ian Atol & Joe Emenaker

Contents

Compiler Overview

Parsing

We used a library (serde_json) to generate an AST of our own datatypes from the JSON given by the included ANTLR parser. While we experimented with creating our own ANTLR parser, as well as with making a lexer and parser from scratch, we ended up choosing the use the provided one. Compared to the time perfecting a lexer/parser compared, it seemed much more time efficient to build a way to process an JSON representing an AST that we know is correct. Our parser, then, is a fairly straight-forward construction of our AST by matching JSON objects to their corresponding syntax tree.

Our AST data structure is made of simple expressions, statements, functions, and atomic types corresponding closely to those outlined in the grammar for the language. We did have to expand upon these types at later points for technical reasons. For example, many LLVM instructions ask for a value of type i8. To differentiate values of this type from normal, 32-bit integers, we introduced a "SmallInt" type. Similarly, though the grammar doesn't specifically mention pointers, the specification required that we represent structs using pointers. As such, we extended our types to capture this implicit requirement. However, these extensions to the language's types are only for use with other parts of the compiler and not used by the parser, since valid mini programs only contain the types mentioned in the grammar.

Static Semantics

Our type checker enforces the rules of a simple type system logically derived from mini's grammar as well as described by the specification. One example of this is checking that a called function's arguments match it's declared parameter types, and that each return path returns a type of the declared return type.

The checker works by first populating a context from the function, variable, and struct declarations (and checking for redeclarations per the specification). Then, proceed through the AST, ensuring that each statement, and any nested statements, expressions, and function calls within preserve type correctness. Any error found will halt the type checking process and return the appropriate error.

Since mini only supports 3 types and doesn't involve subtyping or other more complex type concepts, creating the type checker was relatively straightforward.

Intermediate Representation

For the most part, breaking the AST into a control-flow graph was uneventful. Structurally, we represented a function as a set of CFBlock types which had a hashtable of phis, a vector of Instruction types, and a BlockTerminator (which could be either Return, UncondJump{label}, or CondJump{expr, true_label, false_label}).

The only part we had trouble getting right was the phis.

Building Phis

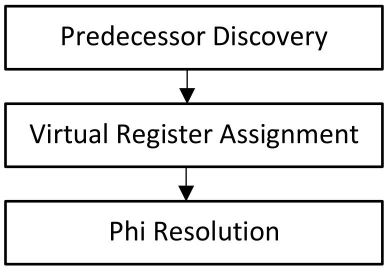
We ended up modifying Braun's method for building phis in our SSA implementation. Although certainly slower, we believe it to be safer, especially for first-time compiler authors. After carefully reading their paper, it sounds like they are *combining the process of predecessor discovery and virtual register assignment into the same pass*. We infer this from a few statements in particular. At the start of section

2.3, they write (emphasis, ours): "*We call a basic block sealed if no further predecessors will be added to the block. As* *<u>only filled blocks may have successors</u>*, *predecessors are always filled. Note that a sealed block is not necessarily filled. Intuitively, a* *<u>filled block contains all its instructions and can provide variable definitions for its successors</u>*. *Conversely, a sealed block may look up variable definitions in its predecessors as all predecessors are known."*. From this, it seems clear that Braun means for all source variables of a given block to have assigned virtual registers *prior to* following the branch(es) at the end to visit successors.

That's not an overly complicated endeavor, but it requires the compiler to retain some knowledge of the type of each block so that it knows when it can try to resolve the phis (eg. The guard of a 'while' block and the exit block of an 'if' can both consider themselves to be sealed after their second predecessor is added, while the exit/return block of the function should be evaluated last and consider itself to be sealed at that time). To us, this seemed likely to add brittleness into the phi resolution process. Assigning the wrong block type to a while-guard or an if-exit or failing to process the return block last would probably crash the compiler or cause broken phis.

To be safer, we decided to just have one block type, and we could process them in any order, in three separate passes:

Predecessor Discovery
Before analysis of any of a block's instructions, we first discovered all predecessors (which we actually did while creating the blocks, themselves).

Virtual Register Assignment


We went through all of the blocks, again, processing all of the instructions within each block. Whenever we found a reference to a source variable which *was not in the block's local symbol table*, we would add an 'incomplete phi' for it in the block's phi table. Although it's not clear, in Braun's paper, if they mean 'partially complete' when they say incomplete, *ours are totally empty of any predecessor virtual register information*. They are just a vector of (block_id, Option::None) pairs. In other words, we don't even bother trying to distinguish which blocks are 'filled', at this point. In terms of Braun's pseudocode, our `addPhiOperands()` does *not* call `readVariable()` on its predecessors in this pass.

Phi Resolution
Lastly, we iterate through all blocks one more time. We now know that *every* source variable assignment in every block has a virtual register in its symbol table, at this point, so we are safe to iterate through the blocks, and each of their phis, asking each predecessor to recursively look up the source variable.

We found this method to be much easier to debug, as the expense of some compiler speed.

Optimizations
Overall, we had four optional optimizations:

1. **Empty-block/redundant-jump removal**:
   The CFG was scanned for pairs of blocks related by being each others sole

predecessor/successor. In this case, we were able to combine the two blocks' instructions into a single block and remove a branch instruction. Additionally, for blocks which had no instructions *and* didn't have *both* multiple successors *and* a predecessor with multiple successors, we could remove them via the following process:

   a. If the empty block has a single successor, replace *this* block's label with the label of its successor in all predecessor branch instructions
   b. If the empty block has two successors, make sure that all predecessors have single successors and, then, replace all predecessor branches with *this* block's branch

2. **Trivial-phi removal**:
   In accordance with Braun's paper, provided that all virtual registers in a phi were either the virtual register the phi was being assigned to and *one* other virtual register or value, all occurrences of the former can be replaced by the later. This step was placed *after* constant-propagation due to the possibility of multiple virtual registers coalescing to the same constant value.
3. **Dead-code removal**:
   We used the mark-and-sweep method to iteratively remove code which assigned to virtual registers which were never used.
4. **Constant-propagation**:
   Sparse-simple constant propagation was used

When all of these optimizations were applied, we saw a 24% reduction in the size of our register-based ARM code, an 20% reduction in the size of the binaries (aggregate for the 20 benchmarks) and an 11% reduction in the execution times (of the 4 benchmarks which ran successfully *and* had appreciable execution times).

Code Generation and Register Allocation

Code generation was fairly straightforward once we converted to using an 'Instruction' enum which had variants for all of the LLVM instructions we would need. This Instruction type then had as_llvm() and as_arm() methods which we could call on each instruction in a block to generate the LLVM or ARM instructions. For the as_llvm() method, each instruction usually generated a single LLVM instruction. For the as_arm() method, it could generate up to six instructions like with a Compare between two spilled registers:

```
mov lr, #0 @ Store a 'false' in our result register
ldr r12, [fp, #-12] @ Load first spilled register
ldr r10, [fp, #-20] @ Load second spilled register
cmp r12, r10 @ Compare them
moveq lr, #1 @ Set result to 'true'
str lr, [fp, #-16] @ Store the result in another spilled register
```

Moving values between globals, stack, and registers

ARM's 'mov' instruction has some limitations in the types of operands it can use. Although the *source* can be a "flexible second operand" (a register, an immediate, or memory location – indexed or not), the *destination* must be a register.

Because of this, our compiler couldn't just naively output 'mov' instructions for any virtual register's "real" location, since the destination could be a place in global memory (for global variables), the stack (for spilled local variables), or real registers.

Our solution was to create a "mov()" function which accepted two ARMOperand types and a scratch register to use, if needed (I.e. for moving memory to memory). The ARMOperand was an enum which could represent immediate values, global memory locations, stack offsets (for spilled registers), or registers. Based upon the combination of destination and source, mov() could output either a single mov instruction (when one of the operands was a register) or, in the case of memory-to-memory, two mov's, using the scratch register to hold the value between instructons. One small optimization introduced, here, was that mov's with an immediate value less than 256 could use mov, while ones between 256 and 65535 could use a single movw, and those over 65535 would use a movw/movt pair.

If we had time, we definitely would have expanded this technique to include some other instructions like cmp, add, sub, mul. Because these also restrict the first operand to be a register, our compiler *did* naively use its mov() function to ensure that the first ARMOperand argument was definitely available in a register, causing useless mov instructions when the first operand was *already* a register.

Nasty Bugs

We encountered a few very interesting bugs, some of them rather hard to track down. We mention them, here, in the hopes of saving the time of later generations.

Order of ARM instructions in compare operations

The method, given in class, for converting a icmp LLVM instruction like:

```
%v3 = icmp eq i32 %v1, %v2
```

Is to mov a 0 into %v3, perform a cmp on %v1 and %v2, and then perform a moveq to conditionally move a 1 into %v3:

```
mov %v3, #0
cmp %v1, %v2
moveq %v3, #1
```

Keep in mind that we did our register assignment based upon the *intermediate representation*, not the ARM representation. Therefore, if this instruction were the last one making use of %v1 or %v2, this would mark the end of their live range, and their real register would be available for assignment to %v3. So, we can see what happens when a variable gets reused... for example, when %v1=r1 and %v2=r2 and they both reach the end of their live range and r1 is given to %v3:

```
mov r1, #0 @ %v1's value is lost before we can use it!
cmp r1, r2
moveq r1, #1
```

There were two possible solutions that we could see:

1. Use a scratch register to hold the 0 and 1 result for %v3 until the end of the multiple ARM instructions generated for this single intermediate instruction (this is what we did, but at the cost of a *third* scratch register, since we needed two to hold spilled registers for the cmp), or

2. Delay register assignment until ARM instructions are produced. This probably would have solved the issue (and also solved a problem we had with phis, mentioned next), but it brings up a new problem. We would need a way of representing ARM instructions with *virtual* registers or with *real* registers. If we were to try this approach, we would probably leverage generics to have something like Vec<ARMInstruction<VirtualRegister>> and Vec<ARMInstruction<ARMOperand>> (where ARMOperand could be one of the various things a virtual register could turn out to be... a real register, a stack offset, an immediate).

Order of mov instructions in phi processing

When represented in assembly, the phis are turned into a series of mov operations moving values from the registers holding them in the *current* block into the registers which will hold them in the *next* block. We ran into a problem, however, with the *order* these moves are done in, since one real register can get a new value before we're done using it. For example,

```
block1:
   …
   br label %block2
block2:
   %v3 = phi [block1, %v1]…
   %v4 = phi [block1, %v2]…
```

And now, let's suppose that %v1 and %v4 get assigned r0 and %v2 and %v3 get assigned r1. We end up with the following ARM code:

```
block1:
   …
   mov r1, r0 @ %v3 <- %v1
   mov r0, r1 @ %v4 <- %v2 (%v2's value is already lost!)
   b block2
block2:
   %v3 = phi [block1, %v1]…
   %v4 = phi [block1, %v2]…
```

Our hunch is that this is due to phis being in a hashtable, and their order of processing during live-variable analysis being different from their order of processing at conversion to ARM. As with the previous problem, there are a few solutions which come to mind; one simple yet inefficient, the others better but harder to implement:

1. We could simply push all of our 'source' registers to the stack and pop the values to the 'destination' registers in reverse order (what we ended up doing)
2. Make sure that we always process them in the same order (we didn't think of this until it was too late)
3. We could perform a dependency analysis on the sources/destinations so that we could do the movs in the proper order (and using a scratch variable if we had a cycle in the dependency graph)

4. Lastly, as with the previous problem, we probably could have avoided this if we had implemented a 'second intermediate representation' representing ARM instructions but using virtual registers and, *then*, performing the register assignment.

Order of mov instructions on argument unpacking

Just as with the previous problem, we had the same issue of data being overwritten when we tried moving the function parameters to their assigned registers in the starting block. For example:

```
define i32 mod(i32 %a, i32 %b)
   @ %a assigned r1
   @ %b assigned r0
```

Clearly, this is going to result in r0 (as arg1) being placed into r1 (for %1) while we still needed r1's old value (as arg2 for %b). We could have tried the push/pop or dependency analysis methods from above, or we could have tried creating a pseudo instruction which killed the four arguments, in order, at the very beginning of the function, but we used a different solution:

During the graph-coloring phase of the register assignment, we *pre-colored* the parameters vars. In other words, the virtual registers corresponding to the *first four* function parameters were assigned registers matching their position (eg. In the above example, %a is assigned r0, and %b is assigned r1) and placed into the color graph *first*. Then, all of the 'function body' registers were added and had to be colored so as not to conflict with the colors already assigned to the parameters. (Note, although we put edges in the graph between all of the parameter registers, that probably wasn't necessary, since we know they're going to be colored differently from each other and the edges between them don't affect how the later registers get colored). This had a side benefit of eliminating some mov's at the start of the function, where we were shuffling values around to different registers.

Analysis

Benchmarks were all performed at the same time on a RaspberryPi 4. Because the pi's will throttle the CPU speed if the processor gets near thermal limits, we ran benchmarks for about 5 minutes before we ran the ones we used for actual numbers.

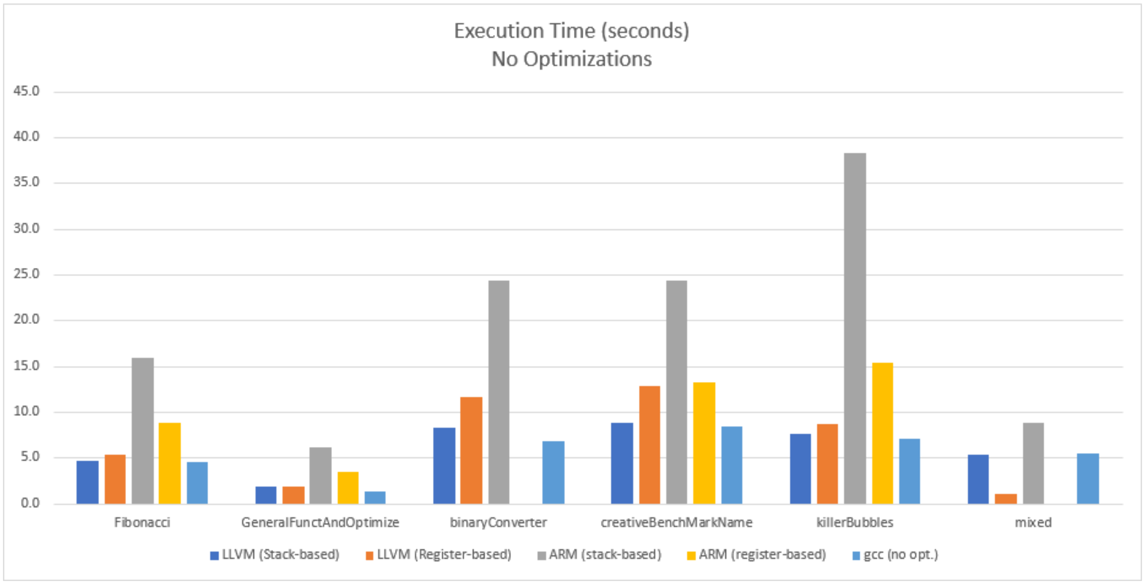
For gauging the effectiveness of optimizations, we recorded execution time, size of source file, and size of binary executable. For some of the benchmarks which complete in just a few seconds, these were left out of the speed measurements. For comparing sizes of source files, most comments were stripped out, and, for comparing sizes of compiled binaries, we used the sizes of the unlinked object files after running `strip` on them.

A few different combinations of optimizations were tested:

- No optimization
- Trivial phi removal only
- Dead code removal only
- Constant propagation only
- All three optimizations, plus empty block removal

Execution Speed

By far, the biggest surprise is that, in register-based ARM mode, our compiler, with no optimizations, only took twice as long as gcc (also with no optimizations). In the chart below, we show execution times for no optimizations for the six long-running benchmarks (register-based ARM is missing in two of these, since its output didn't match the reference output)



This is especially promising since there are some *glaring* inefficiencies in our compiler which could tighten the gap. First of all, there are plenty of redundant mov operations. For example, we have some portions which look like this:

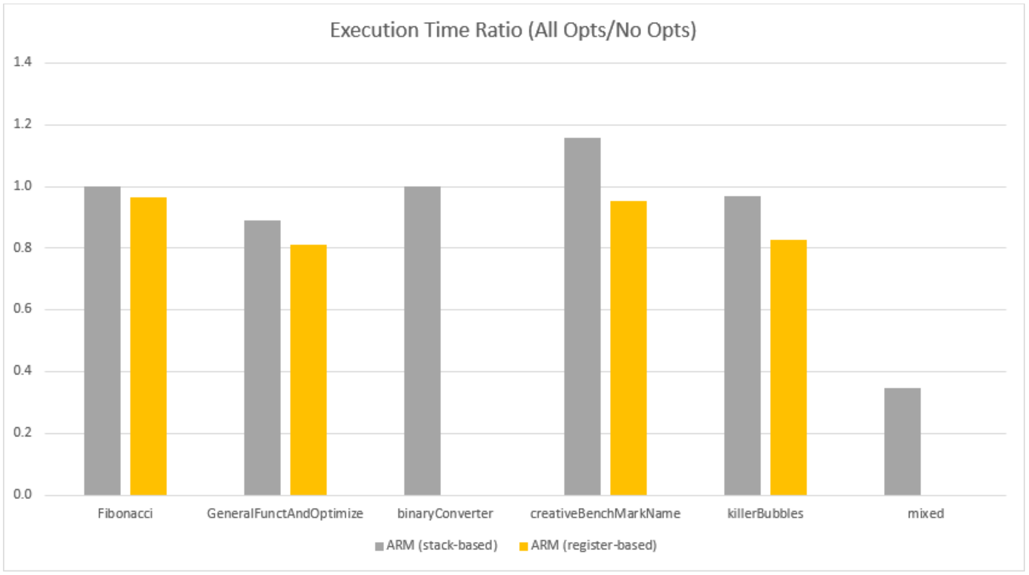
```
mov r0, lr @ lr is one of our scratch registers
mov r0, lr
mov lr, r0
```

The double move from lr to r0 is probably a duplicated line in our source code, somewhere, but the third line is almost certainly the start of a new IR instruction that has no insight about the state of the registers. At the very least, two of these instructions could be removed, and, unless we knew that we needed the value in r0 later, *all* the instructions could be removed.

Before we had moved on to the register assignment, our stack-based compiler actually performed much better against gcc than is shown in our final tables, but, in order to get the register-based ARM working, we had to introduce a lot of precautionary mov's to scratch registers and/or pushes.

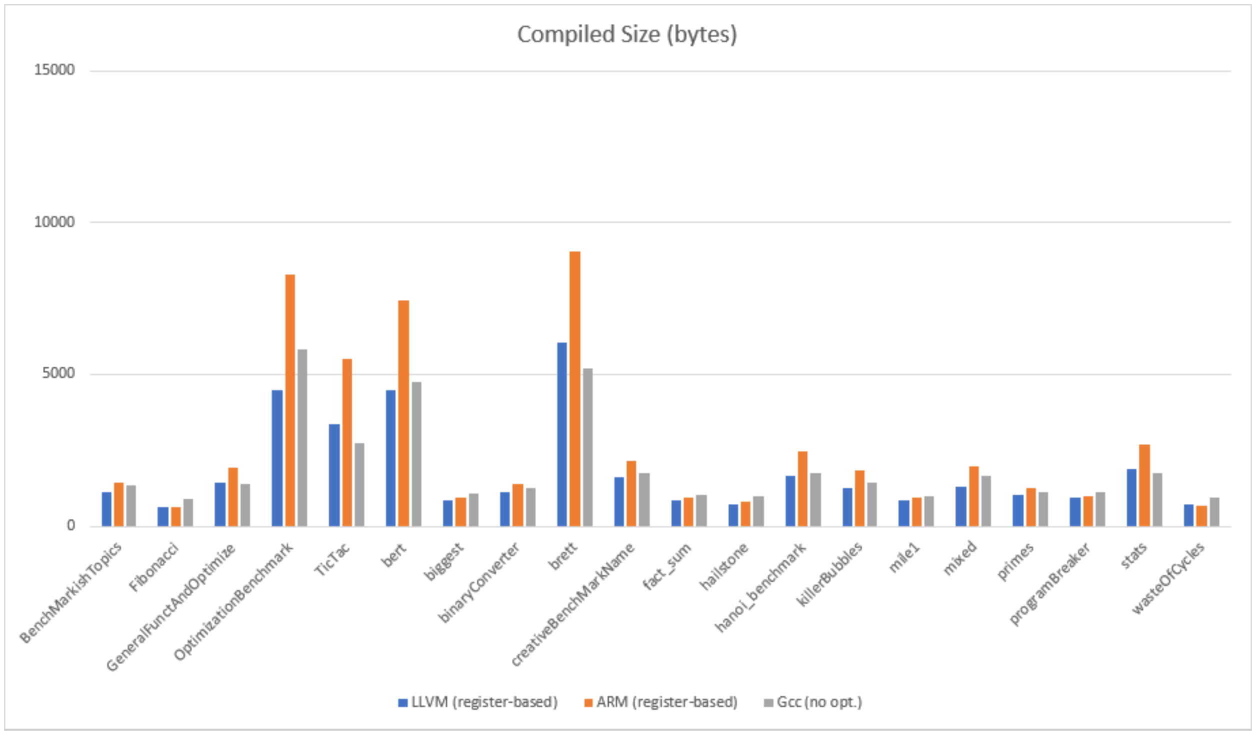
Effect of Optimizations

The effect of optimizations on execution time were a little interesting. The chart below shows the ratio of execution times between unoptimized and fully-optimized code for both stack-based and register based ARM code. In stack-based, we saw very little improvement (except on 'mixed'), while our register-based execution times were between 5%-20% less (for the ones where the output matched the reference).



Binary Size

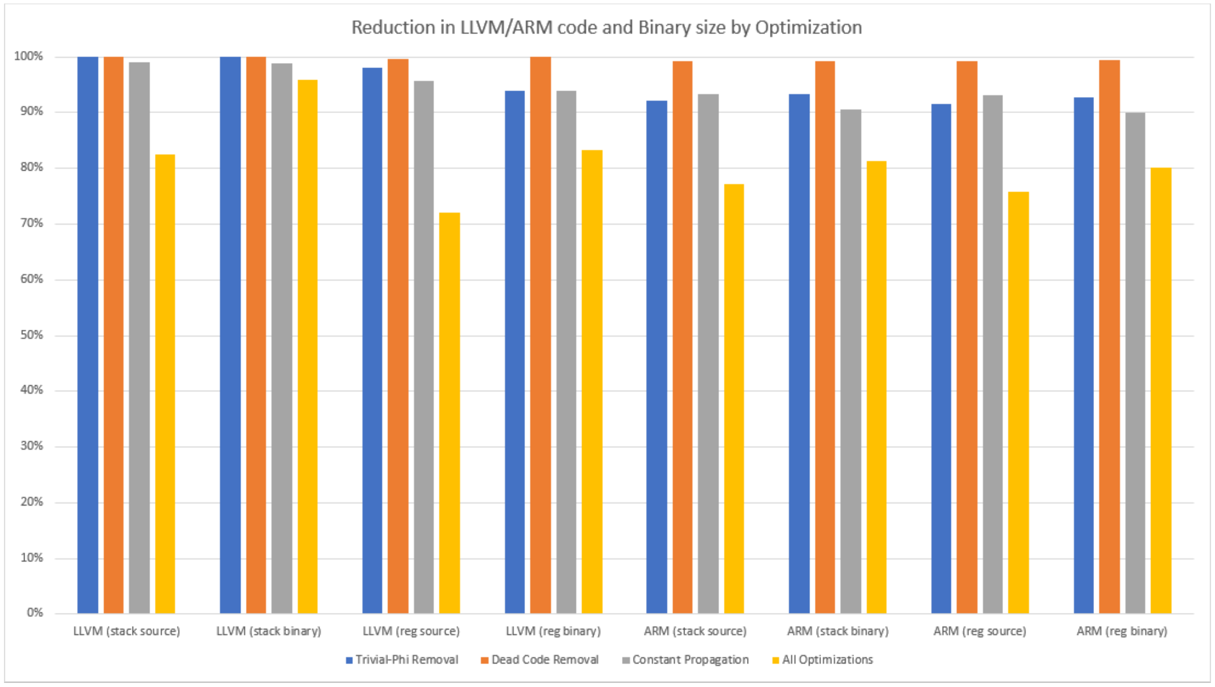
Although size is almost never a large concern in today's computing environments, it *can* give us some insight into how many superfluous instructions we've got (like in the last section). For our size comparisons, we compared the compiled, unlinked, symbol-stripped object files of our register-based LLVM and ARM outputs against unoptimized gcc.



Although there were a few cases where our binary size was *double* that of gcc, most of our binaries were within about 10% of gcc's output (sometimes smaller), and these size differences would be even *smaller* (relative to overall size) if we were fully-linked executables.

Effect of Optimizations

We looked at how each optimization (except empty-block removal) reduced the size of the LLVM/ARM output in proportion to the unoptimized size. We can see that dead-code removal was the least-effective optimization (only shaving about 2% off of the code size) while trivial-phi removal and constant propagation were very similar (reducing between 5-10%, usually). When all of these were combined, we saw some *very* good results, reductions of 20-30%, which we suspect are a synergy between the constant propagation and the trivial phi removal. We made sure to propagate constants *before* trivial phis, and, now, it would have been interesting to reverse their order and run the benchmarks again, to see how much the order matters.



Miscellany

Non-determinism in the compiler

The part of our compiler which produces ARM code iterates over a few hashtables and hashsets (especially during register assignment). For whatever reason, Rust seems to incorporate the system time into the hashing algorithm (or finds some other way for the hashing to vary between executions of the compiler), which caused the register assignments to vary with each compilation (although they still satisfied the interference graph coloring). For development, this was, initially, a hinderance, since we couldn't figure out why tests/benchmarks *sometimes* worked and *sometimes* didn't. Ultimately, however, this was a handy tool for isolating some bugs which, otherwise, could have taken much longer to track down.

By running the compiler several times, we were able to generate two different ARM sources, one which worked and one which didn't. We could then diff these two to find the only places where the sources differed, helping us to focus on a smaller part of the code. Before this, we had tried comparing our ARM source to the output of `clang –S`, but the code was so different, it was almost no help.

Although it proved to be a handy diagnostic tool during development, for a production compiler, we would probably want to add a sort() to all of those iterators to make the compiler completely deterministic.

Notes on using Rust

For all of its new-ish features (optionals, pattern-matching, borrow-checking), Rust doesn't have inheritance like most OO languages. It has *Trait* subtyping (in the way that Java can have interfaces extend other interfaces), but the lack of inheriting object/structure fields and, especially, methods, was a bit of a problem.

One example is how we were forced to handle contextual information about each function (eg. It's ID, arguments, local vars, etc). There were some aspects of this function context that we needed access to at most phases of compilation (eg. the argument and return types were needed by the type-checker and the llvm generator) but there were other parts which were needed only by *single* parts (like the ARM generation needing the gen/kill/LiveIn/LiveOut sets). Rust wouldn't allow us to have a generic FunctionContext which then had LLVMFunctionContext and ARMFunctionContext types.

Another example of how this constrained the design of data structures was in how we handled control-flow blocks. Every control-flow block terminates in one of three ways: an unconditional jump to one other block, a jump to one of *two* other blocks depending upon a condition, or a return. With a typical OO language, we could have made a CFBlock superclass and then UnconditionalJumpCFBlock, ConditionalJumpCFBlock, and ReturnCFBlock subclasses which inherited common behavior. In the case of Rust, we had to create a CFBlock structure which contained a field (an enum type called CFBlockTerminator) describing how it terminated. This, in itself, wouldn't have been much of a problem were it not for the way Rust's borrow-checker guards mutability.

At its core, Rust's borrow checker seems to follow the standard reader/writer resource locking paradigm where you can have many readers *or* one writer. If a piece of data is in scope for writing (mutable), it is not allowed to be assigned to another variable for reading, etc. The way this becomes an issue is because it doesn't track per-record mutability. Here's an example:

```
struct CFBlock {
        stmts: Vec<Statement>  // All source statements for this block
        symbol_table: HashMap<String,String>
        term: CFBlockTerminator // Does it terminate with a branch, a return?
        id: String // what's its name, for branch labels?
}
        …
for block in blocks {
        // Have all statements update the symbol table (assignments, etc)
        for stmt in block.stmts {
                stmt.update_symbol_table(&mut block);
        }
}
```

What happens, here, is that part of the block (stmts) is borrowed for reading, so we're not allowed to pass its containing struct (CFBlock) as a mutable parameter. This was extremely problematic since we were using structures for housing the entire context of the program (global vars, types of all functions, etc) and for the entire context of a function (all control-flow blocks, symbol tables, phis, etc). As one can imagine, there are plenty of times when we would want to iterate over one of those fields (all phi's, say) while calling a method to update another field (the symbol tables, for example). There were three ways to deal with this:

1. Don't pass the containing struct. If iterating over something in the structure (like *block.phis*), don't pass *block*; pass the other fields you need. This we almost never did, since we would have had to pass *so* many of the individual fields, as each function called other functions which needed other parts of the context structure. We would have been passing a dozen things... most of them individual fields of one struct (defeating the whole point of having structs instead of a bunch of individual variables)
2. Clone what you're iterating over. This was the preferred approach, especially if we were just iterating over the keys to a hashtable (although Rust forces you to clone the entire hashtable and take the keys from *that*, otherwise, it thinks you borrowed the original because keys() is a function, and Rust loses granular tracking of borrow when passed as function parameters... even as *&self*). Certainly, thought should be given as to what the hashtable is containing. Cloning a hashtable of the control-flow structure would have cloned every function, all statements, variables, etc. Care must also be taken to not make the mistake of mutating the soon-to-be-discarded clone, or your changes won't take effect.
3. Make a to-do list. In a few cases (in eliminating trivial phis, we needed to examine all instructions for instances of the virtual register we were eliminating to replace them with the virtual register we were replacing it with), cloning wasn't an option because we needed to be able to mutate what we were iterating over (the instructions) but the instruction.replace_register() method needed to be able to mutate something like the symbol tables (which were in the same structure as the instructions). What we ultimately had to do was make a vector of changes which needed to made to statements, and *after that loop*, we had to loop over the vector of changes and apply them.

Lessons Learned

- For an implementation language, select a language you either already have a parser for (or can create a parser for quickly) or are very comfortable reading JSON in. The course does now allow a lot of free time to experiment with an unfamiliar parsing library or to learn how to import JSON into your AST data structures.
- Have an actual type for LLVM instructions. Originally, eager to get something which would compile in LLVM, we produced the instructions as strings. Although, in lecture, it was mentioned that it was advisable to have an IR class/type, we had no idea just how crucial it was to do this, for the following reasons:

    1. Generation of ARM code is done, almost directly, from the LLVM instructions. Although this can lead to some inefficient ARM code[1], just getting the LLVM code to work (especially with structs) is such a delicate process, that it is a wise choice to make the generation of ARM code a very mechanical one, where each LLVM instruction generates one or a handful of ARM statements which, crucially, are practically just a textual transformation, requiring a minimum of extra analysis.
    2. Many of the optimization steps (although introduced at the end of the course) are applied to the IR, so it is important to not have the semantic meanings of the instructions (eg. which register is being defined, which registers are being used, whether an operand is a local/global/immediate) lost due to conversion to text too early.
- Have a reliable way of addressing/replacing/deleting instructions. Many of the optimization steps involved an analysis step which identified possible instruction changes/removals, and then a step to actually do them. The first step needs a way to indicate the instruction needing removal/change to the second. With most languages, you could probably just have a pointer to each one, but we were afraid of getting into trouble with Rust's Rc's and WeakRc's. Because of this, and because each set of instructions in each CFG block is just a vector, we decided to make an InstructionReference type which contained a CFG block_id and the index of the instruction in that block's vector of instructions.
- We probably should have made phis normal instructions. Because phis and branches always go at the beginning and end of a block, respectively, it seemed prudent to have our CFG objects keep them separate (phis, instructions, and the ending branch) so that they could be processed in a way particular to their nature (eg. we don't need to check getelementptr for being a trivial phi). Also, it seemed to make sense to keep phis in a hashtable (to ensure that we never ended up with multiple phis for the same virtual register, for example), while instructions were in a vector.

  That was probably a mistake.

  As the project went on, this ended up being quite a liability, as we had to pass two different data structures (branches *were* turned into a subtype of Instruction) to anything that needed to scan for register uses/assignments (eg. dead code removal, constant propagation, register assignment, etc.). Even worse, any function/method which needed to iterate over anything which defined/used a register needed *two* for-loops, one which did the phis and one which did the "regular" instructions, and, in a function which needed to go through the instructions *backward*, the juxtaposition of these two loops needed to be reversed, which either needed a second do-them-in-reverse function or a single function would need the phi for-loop before and after the Instruction for-loop, with a boolean flag to determine which one executed. It ended up being a very confusing, bloated mess... all out of fear that some phis could end up down among other instructions or that two phis could end up trying to assign to the same virtual register.

[1] There are probably some sequences of LLVM code which could be represented by fewer ARM instructions than are generated by our method of blindly generating ARM from each LLVM, ignorant of the instructions before/after it, but these could probably be fixed if we, like we did with LLVM, created a class/type which represented ARM instructions so that we could more easily perform analysis on the ARM code before it got turned into text.

As just mentioned, we *did* turn the block-terminating branches into instructions (because there was always only one and they didn't assign to anything). Before we did that, however, iterating over a block required *three* steps: the phis, the instructions, then the terminator.

Future students would be well served being told not to make such a mistake. By the time you're ready to start outputting your LLVM code, you probably should have everything in a hashtable or graph of arrays/vectors of some kind of Instruction type. Removal of trivial phis, then, could just leverage whatever methods already exist for dead-code removal and constant-propagation.

Other optimization ideas - Things which we'd implement, if we had more time

Economizing on stack operations when calling a function:

> Of the registers in r0-r3, only push the ones which are actually assigned.
>
> After our register assignments are done, if we have some real registers left over, we could mov some of our r0-r3 to the spares to avoid needing to push them before a function call. In fact, simply by prioritizing registers r4-r11, we could end up with r0-r3 unassigned, thereby avoiding needing to push/pop them before/after a function call.

Economizing on stack ops when being called:

> For functions which never call another function, the lr register is never mutated by any bl calls, so there would be no need to push it.
>
> For function arguments which are spilled to the stack, a naïve implementation would be to copy them from their values from the incoming-argument area (fp *plus* <some offset>) to their "spilled" location (fp *minus* <some other offset>). We could save two mov's involving the stack if we could just use the incoming-argument area as the spilled location.
>
> It might be possible to reduce stack operations in a call by *reversing* the order of preference of register assignments in some functions. For example, if the *calling function* preferred the registers r4 onward, then it might not need to push/pop any/some of the lower registers before/after the call. Likewise, if the *called function* preferred r0-r3, it could avoid pushing some of the r4- registers, too.

Giving source variables preference for a single register

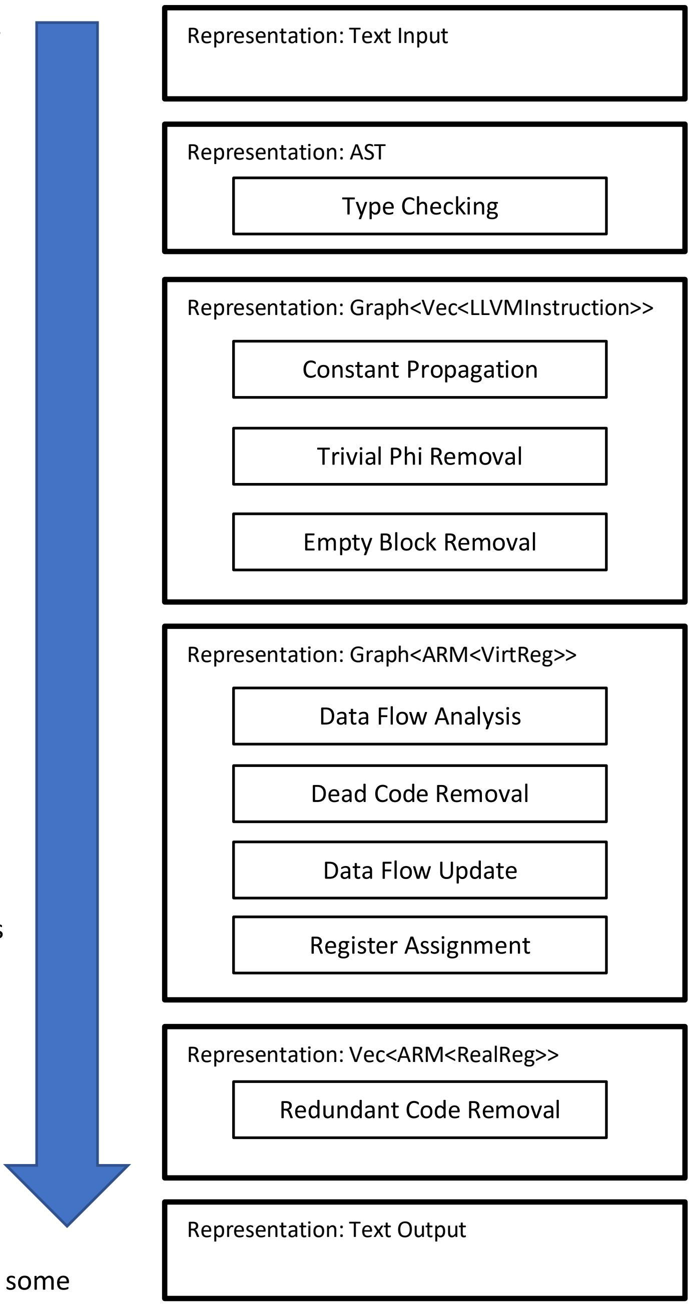
> Our register-assignment procedure was rather naïve. We had a standard order we tried to assign registers in during the "coloring" stage, and we assigned the first one available. This resulted in a lot of mov instructions at the ends of blocks in order to resolve the phi instructions as, say, variable 'limit' was being stored in r0, then in r3, then in r2, and back to r0. Much of this shuffling could have been eliminated if we could mark certain virtual registers as having a 'preferred' register (based upon the real registers already assigned to other virtual registers our current one is moved to/from).

Propagating constants into branching

We didn't have time to allow for propagated constants to propagate all the way into conditional branches. In addition to speeding up the code due to fewer comparisons, it also would have shrunk the executable, as whole blocks could have been pruned.

Conclusions

Aside from the sense of accomplishment of writing our first compiler, the biggest take-away we're left with is the importance of having a variety of representations, each germane to the type of analysis being performed (we shudder to think about how many hours we would have saved – in tracking down and then working around some bugs related to loss of values due to register mov's - if we had only had a representation of the ARM instructions with virtual registers for register assignment). In light of that, we would like to present what our *next* compiler will look like, someday.



While a function is still in control-flow graph form, constant propagation before trivial phis (for reasons already discussed), and then empty block removal, since we may be able to convert some conditional branches into unconditional ones.

Then, we convert the graph to a graph of a type representing the instructions of our target processor, but using virtual registers. This way, more of the assembly instructions are *discretely represented*, so there is less risk that we can accidentally output multiple assembly instructions in an order which destroys some still-needed register data. Note that this will require producing some more virtual registers for this.

After data-flow analysis, dead code removal, and register assignment, we create a vector of ARM instructions with their real register assignments. Here, we can scan for redundant mov's and branches.

Lastly, we produce textual ARM code (which may need some additional ldr/str instructions for any spilled virtual registers.

Done this way, we're confident that we would have a much more reliable compiler producing much faster and smaller code than our first attempt.