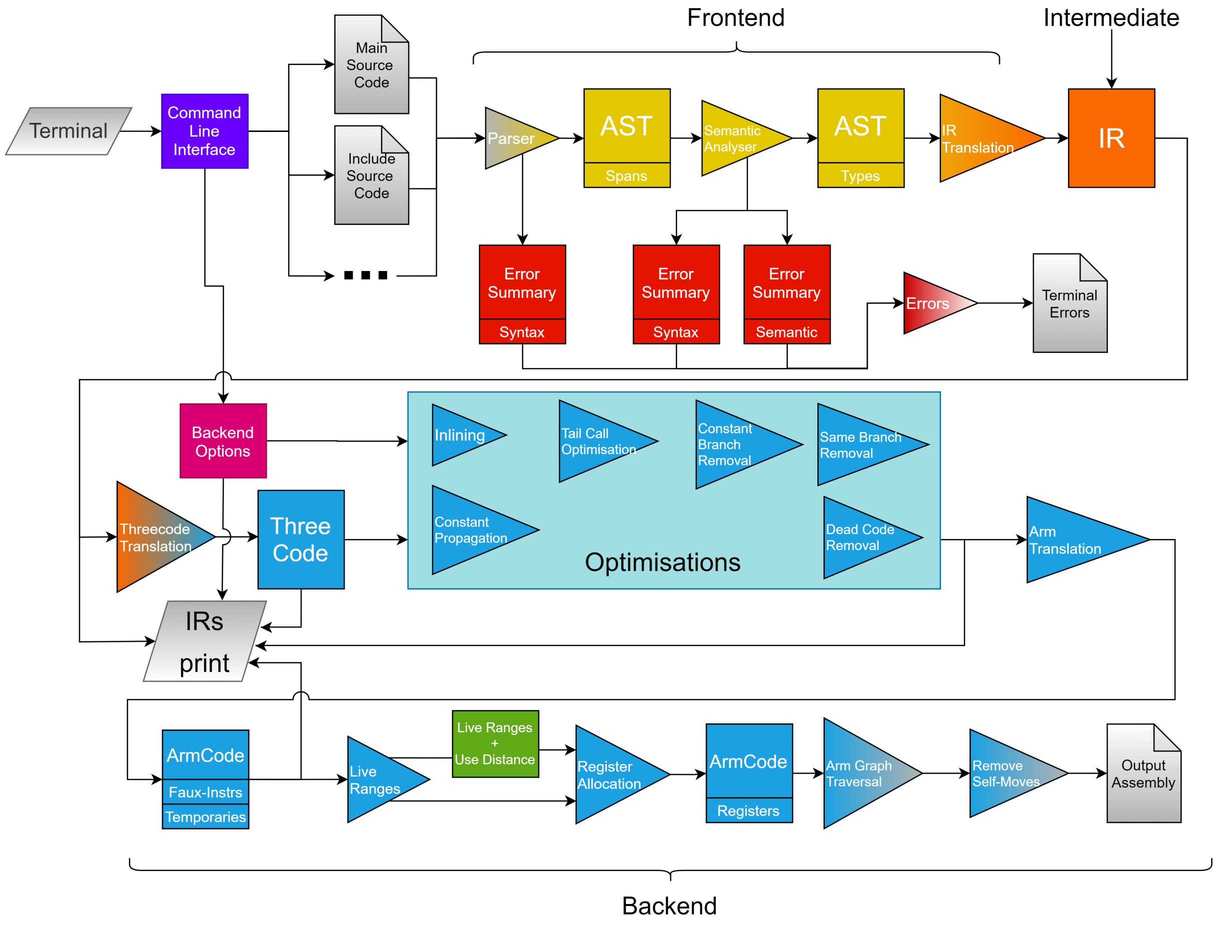
WACC Project Report

WACC 33: Jordan Hall, Bartłomiej Cieślar, Panayiotis Gavriil and Oliver Killane

Compiler Design



Our design maximizes extensibility through modularity on every stage of compilation. Each representation has has an interface and defined semantics, and hence allows for easily swaping out translations, optimisations, entire frontends or translation to architecture specific backends.
The compiler translates the source code through 4 main representations: **AST**, **IR**, **Three-Code** and **Arm-Code**.

AST

The AST is a tree based structure using an extended structure of the **WACC grammar** (allowing for more unary (fst, snd), binary operators (newpair) and types (full pair types)).
Each node in the three is wrapped by a generic. This allows the **AST** to be wrapped with different types of information, without having to change the structure.
Parsing of the input file (and included modules) produces an **AST** wrapped by spans (pointer to the section of relevant source code). Following a successful semantic analysis, the **AST** produced is wrapped by types (if one exists for a given node) to allow for easier translation to our **IR**.

IR

The main **Intermediate Representation** consists of a data section, encoded as structs of static expressions (for the translation not to have to much work to do evaluating those) and a code section, which defines the functions, their types, the types for local variables and their code as well as the code of the main program.
The code is encoded as a dataflow graph of blocks of statements.
All statements, block endings and the data section operate in terms of expressions, which come in 3 different types: **Integer**, **Boolean** and **Pointer**.
As for the guiding principles for this representation, it was optimized for the number of frontends and backends able to interface using it and, secondly, the number of static assertions on it ensured by the type system. Thus, adding a lot of simple frontend/backend features is trivial, requiring little to no changes to this representation.

Three-Code

This representation is the main representation of the backend, sitting below all optimizations, being translated to from the main **IR** and being translated into our arm representation. It consists of a data section, which is a series of data section fields such as strings or integer values, code, represented as a dataflow graph with statements taking at most 2 operands (unless it is a function call) and assigning a value to at most one variable.

Rather than on types, this representation operates on the sizes of the data stored in variables rather than the types.
This representation is designed to be easy to perform non instruction-level optimizations on it. It is not designed to fully support arbitrary architectures; however, it would be moderately easy.

Arm-Code

A control flow graph based arm representation.

Implementation

Choice of Language

We decided to use Rust for the compiler implementation for the following reasons:

- Rust has a strong and powerful type system, which allowed us to write reliable code efficiently without worrying about null-pointer errors or type-related runtime errors.
- The Rust borrow checker ensured that all code we wrote would be memory-safe at compile-time without needing a garbage collector, bringing us to our next point.
- Parallelization of major compilation steps is trivial in Rust due to the borrow checker, which we achieved by changing `iter()` to `par_iter()`.
- Rust is extremely high-performance, which allowed us to do expensive computations (such as inlining, constant propagation, tail-call optimization, efficient register allocation, dead-code removal and same-branch analysis) quickly.
- Rust has many language features you would want from a modern language, such as exhaustive pattern matching, constant generics, closures and higher-order functions, first-class iterators, powerful error handling systems and a trait-based type system. These features allowed the produced code to be concise, reliable, and well-structured.
- Rust's development tools are mature, which gave us easy access to unit testing, benchmarking, linting, formatting, and a seamless package manager for dependencies.

Design Considerations

Where possible, we wrote code in a functional-style. This allowed us to have confidence that the code we wrote didn't have unexpected side effects. We also chose to use iterators where possible because this opens up the possibility of using parallel iterators, various optimizations by the Rust compiler, and makes code easier to read.

External Libraries

A small list of most notable libraries we used in our implementation:

- Nom - a fast parser combinator library used in the parser.
- Nom-supreme - an extension to Nom that gives us error trees for better error handling.
- Clap - a powerful command line parser we used to pass arguments into the compiler.
- Rstest - a test harness extension to cargo's test harness, giving us parameterized testing.
- Rayon - parallelization library that gives us drop-in replacements for parallelized iterators.

Compiler Extensions

We implemented a number of extensions, which we will go through and explain the implementation of:

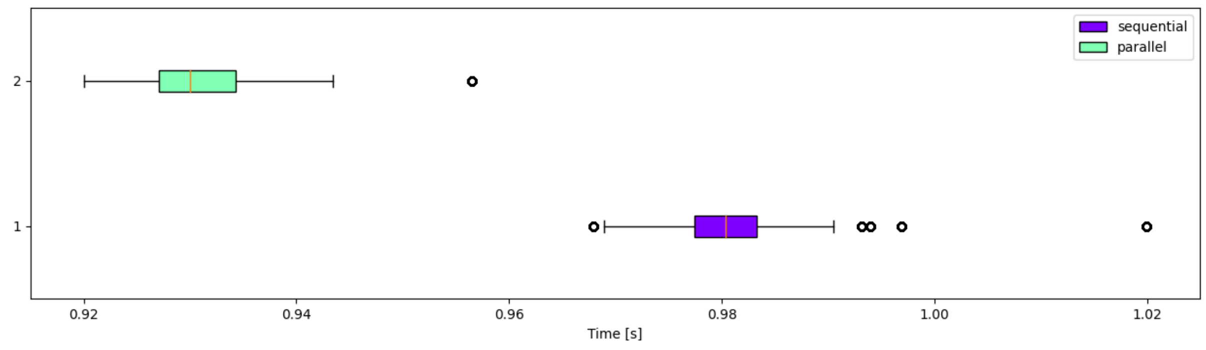
Parallel Compilation

Our compiler performs a lot of expensive optimizations also mentioned in this list. In order to efficiently make use of the fact that most processors have multiple cores, we decided to perform all the major compilation steps in parallel. In total, we have the following steps parallelized:

1. Parsing of modules.
2. Semantic analysis.
3. Conversion from AST to IR over functions.
4. Every optimization on Three-Code.
5. Translation from Three-Code to the Arm Representation.
6. Live range analysis
7. Register allocation

In each of these cases, we iterated over all the functions to perform these optimizations. This was a purposeful design choice because iterators are very easy to parallelize in Rust. Therefore, in most places, we were able to substitute the regular iterators with Rayon's parallel iterators and then collect the data produced by them.

Parallelization yielded an 1.05 ± 0.01x speed-up when compiling tictactoe.wacc with all optimisations on:



Void Functions

We implemented void functions and void function calls to help supplement the standard library. We have also allowed regular (returning) functions to be called without capturing the value as well as the right-hand-side operand. The new syntax is pretty straightforward:

```
begin
    void print_max(int num1, int num2) is
        if num1 > num2 then
            println num1
        else
            println num2
        fi ;
        return
    end

    int print_and_get_increment(int i) is
        i = i + 1 ;
        println i ;
        return i
    end

    call print_max(69, 420);
    call print_and_get_increment(41)
end
```

Notice that we can call non-void functions as well. In this case, we discard the returned value. Calls to void functions cannot be a part of a declaration or assignment statement.

Full Pair Types

Our compiler now supports full pair types, keeping all the semantic information about the nested types. This required a change to the parser, as our semantic analyser supported full pair types during the frontend milestone. We still allow users to write code using the old pairs and so the new feature is also backwards compatible.

Modules/includes

The extended compiler now supports modules and includes through the following syntax:

```
mod ../tic_tac_toe_ai.wacc;
```

Modules themselves can only have functions, and the syntax looks like:

```
mod ../another_include.wacc;
begin
    int add_one(int x) is
        return x + 1
    end
    # ...
end
```

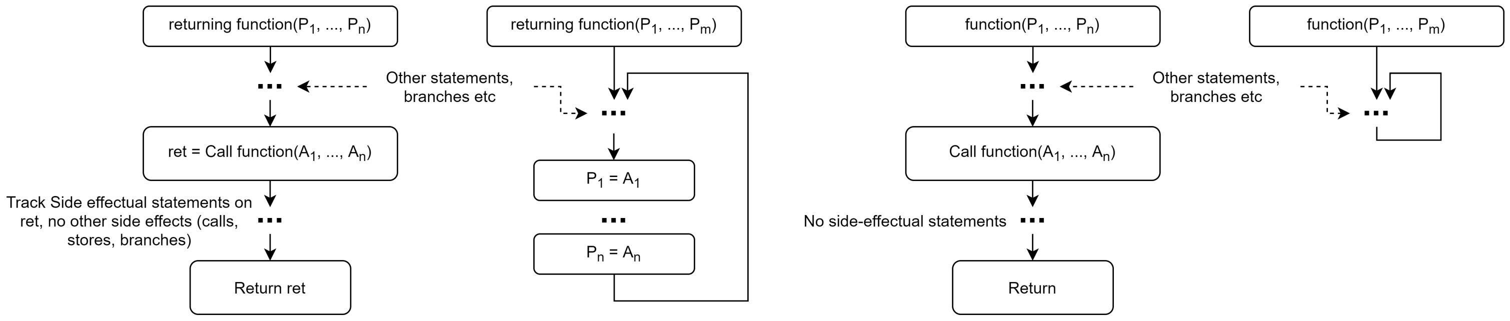
With the "begin .. end" clause being optional.
The way we manage this is by first parsing the beginning of the main file, collecting the locations of each of these module files. We then parse each of these modules individually along with the main file. We then put the module functions into the main file's AST, to be processed as a whole by the rest of the compiler. It is also worth noting that we automatically resolve circular module imports.
This extension also integrates nicely with error systems and is able to still provide reference to the location of the definition for which the error has occurred, thus preserving the original look of the error system.

Function Inlining

We inline calls to functions up to a certain size, which can be set via one of four settings: off, low, medium, and high. We inline functions up to a certain point, determined by a heuristic for the number of instructions that the representation will be translated to. We then perform a breadth-first-search through the functions of the program and choose to expand them based on this heuristic. Inlining has the advantage of not requiring a new stack frame when an inlined function executes, improving speed of program execution through cache locality, and when coupled with other optimizations reducing program size. Inlining becomes more powerful when using constant propagation and dead code removal, as it provides more information for those optimizations. The algorithm is linear in terms of the size of the final code produced. That does mean, however, that it may cause other optimizations such as efficient register allocation to run slower, since there is much more temporary variables to optimize on.

Tail Call Optimization

Tail call optimisation to convert uses of unmodified results from recursive calls, or void recursive calls with no side-effectual code between the call and the function return. This optimisation enables infinite recursion, and arbitrarily deep recursion in many programs. It is improved when combined with inlining (to remove calls which block tail call optimisation).



returning function(P_1, ..., P_n)
Other statements, branches etc
ret = Call function(A_1, ..., A_n)
Track Side effectual statements on ret, no other side effects (calls, stores, branches)
Return ret
returning function(P_1, ..., P_m)
P_1 = A_1
P_n = A_n
function(P_1, ..., P_n)
Other statements, branches etc
Call function(A_1, ..., A_n)
No side-effectual statements
Return
function(P_1, ..., P_m)

Efficient Register Allocation

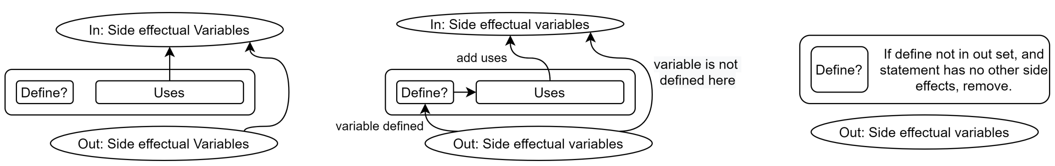
Registers are allocated using live ranges (combined with use-distance calculation), and a temporary based arm with some special instructions (call, assign stack word, return). Once live ranges are calculated, each function is traversed with an allocation state (determining what values are in registers, tracking the stack pointer position relative to the start of the subroutine call and the stack frame layout. All arm instructions have their temporaries translated to registers, and instructions to manage spilling to and from the stack frame inserted, as well as to manage the arm calling convention.
As live ranges are calculated on a per-instruction basis, in a single statement the destination register assigned can be the same as some of the argument registers (if they are not live afterwards), furthermore preserved registers are only spilled when necessary. Spilling of registers holding temporary values is only done when no preserved values can be spilled, or free registers available. The temporary next used furthest away is spilled to keep frequently/soon to be used temporaries in registers.

Constant Propagation

This optimization pre-computes the values for the variable assignments. It works by first computing the live variables at each three code statement in order to optimize the followng step, and secondly, performing a dataflow analysis of the definitions arriving at each use of each variable. This is then used to iteratively optimize statements by having them flagged as ”updated” and then possibly flagging the nodes that use the variable defined in that statement as ”updated” as well. This means that, apart from the dataflow analysis, which can easily be cached by moving to an SSA representation, the rest of the algorithm is linear in complexity.

Dead Code Removal

Removal of statements that have no effect on memory, returned values or control flow. Particularly useful when combined with inlining. Statement liveness is determined using our dataflow analysis system to construct live in and out sets of side-effectual variables.



In: Side effectual Variables
Define?
Uses
Out: Side effectual Variables
In: Side effectual variables
add uses
variable is not defined here
Define?
Uses
variable defined
Out: Side effectual variables
Define?
If define not in out set, and statement has no other side effects, remove.
Out: Side effectual variables

Same Branch Optimization

This optimization removes conditional branches that have both branches lead to directly the same statement. Helps clean up the code after the optimizations, and even enables better performance for others (e.g. Dead Code Removal). The optimization works by first scanning all the nodes that we can possibly optimize away in this way, and then flagging potential new candidates for an optimization like that. This optimization is linear in complexity and thus can be used in any place in the optimization stack.

Constant Branch Optimization

This optimization removes conditional branches whenever the condition is based on a known constant. Works best with constant propagation, as it can optimize the expressions for calculating the constant down. This optimization is linear in complexity and thus can be used in any place in the optimization stack.

Standard Library

To aid the usability of the WACC language, we devised and implemented a standard library in WACC. This standard library can be imported and used elsewhere. The library is complete with the following functions:

For common array functions, we implemented the following for Ints, Bools, and Characters: printing, fill, count last, find nth, find first, find last, binary search, contains, swap, copy, move (with reverse and free), concatenation, fold with AND/OR, bubble sort, heap sort, and quick sort.

For common math expressions, we implemented the following: min, max, abs, pow, truncated square root, truncated log2, signed sin, signed cos, signed tan, sum/product over Int arrays, truncated mean, truncated population variance, truncated population standard deviation, truncated sample variance, and truncated sample standard deviation.

We also implemented a python test generator for each of the functions in our standard library, and implemented them as a part of our test suite. This is an interactive test generator, and so we are able to generate tests for different types and quickly test our standard library

Project Management

What went well

During the course of this lab, we made sure to set up good communication channels to keep the group synchronized and well-planned. For centralized communication, we set up a Discord server. We also attended weekly meetings, where we discussed what we had done, what we had to do, and divided tasks among the group.

We created a branch for each major compiler feature. Coupled with a strict CI compliance, code reviews before merge, and the protected master branch, this ensured that changes to the master branch would not corrupt previous changes, and not make the master branch fail any tests it had in its test suite.

Furthermore, the CI had 4 stages. These included: building, testing, linting, documentation reports, and deploying the documentation to GitLab pages. These proved sufficient for keeping the master branch at a high standard of code hygiene.

Finally, we hosted many group-programming sessions in person. The high channel of communication and pair-programming allowed us to accelerate the speed at which we programmed, and relieving stress among the group mates.

What could be improved

During the backend milestone, we overscoped what we were expecting from our compiler. This caused the final product not to be finished until after the backend deadline. To improve this, we would have made a simplified, working version of the compiler early on in the milestone and iterated over that design until we had our finished product. This would ensure that we have a working product by the submission deadline.

Final Product and Possible Future Improvements

To ensure functional correctness, we tested extensively with ≈ 1600 tests (unit, integration tests with combinations of optimisations). We also have print options for the IR, threecode, and armcode (with temporaries) available in the compiler command line interface so that we can check how optimisations are being performed as expected.

We designed our compiler to be as extensible as possible, with separated modules and well defined interfaces for the frontend, intermediate and separation in the backend between the architecture specifc arm representation, and the non-specific/general threecode. This allows us to easily add new features to the wacc grammar and semantic analyser without requiring any code changes in the backend, or to implement new backends without altering any of the frontend or threecode (thus allowing all existing threecode optimisations to be used).

Furthermore the use of a common graph, dataflow analysis tools makes implementing new optimisations, and assembly representations much easier.

We spent much time on the user experience. Our command line interface is feature full, and includes flags for viewing the intermdiate representation, threecode before and after optimisations as well as the arm representation with temporaries. Furthermore the documentation (as published to gitlab pages by our CI) is extensive, covering all modules, and expected behaviours (e.g CLI options, frontend exit codes, etc).

Some possible future improvements we could make to our compiler to take advantage of the current extensible structure are:

- Adding an x86 backend, by translating from our three-code into a separate x86 instruction representation.
- Use LLVM as a backend, by translating out **IR** to it.
- Splitting ARM stack and register allocation to allow for smaller stack frames. We currently do these in one pass, so cannot determine which temporaries will be spilled prior. As a result, all temporaries used need a slot in the stack frame.
- Add an SSA intermediate representation to improve the efficiency of our dataflow analysis algorithms. We did not do that here because of the limited timeframe and stringent correctness requirements.
- Improve upon the WACC language (would require few to the IR):

    1. Classes with inheritance
    2. Some quality-of-life changes (optional semicolon after the last statement, bitwise operators, variable-sized arrays, for/do-while loops, etc.)
    3. Concurrency
    4. Full heap allocation with pointer arithmetic (would have to be carefully integrated in terms of design with classes)