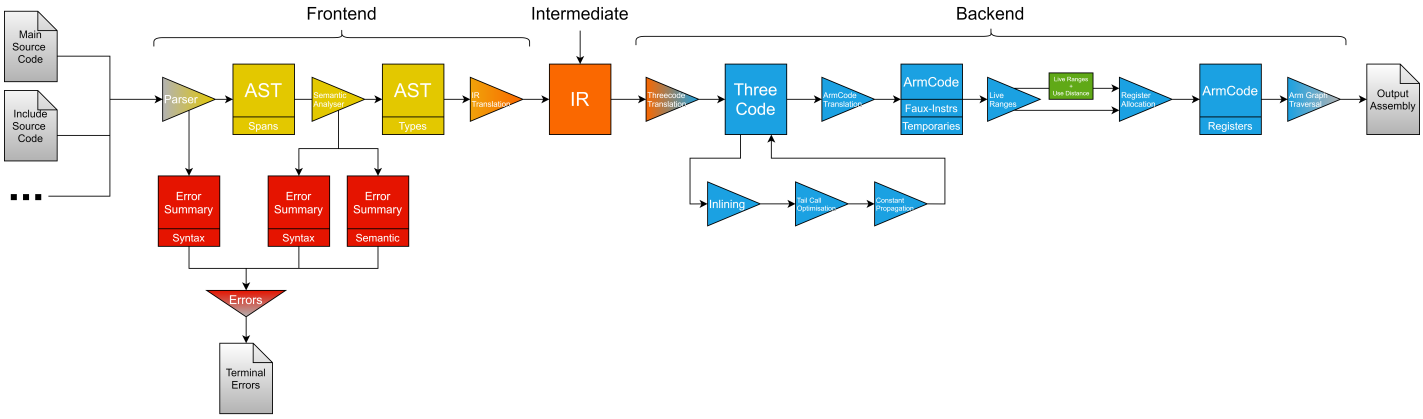
WACC

Jordan Hall, Bartłomiej Cieślar, Panayiotis Gavriil and Oliver Killane

14/03/22

Compiler Design



Our design attempts to maximize extensibility through modularity for every stage of the compiler. Each representation has has an interface and defined semantics, and hence we can easily swap out translations, optimisations, entire frontends or translation to architecture specific backends.

The compiler translates source code through 4 main representations; our **AST**, **IR**, **Three-Code** and **Arm-Code**.

AST

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 intermediate representation consists of a basic data section (for static, immutable data), and functions consisting of block graphs of basic statements.

Expressions can be nested, and come in three main types; **Pointer**, **Number** and **Boolean**.

The semantics of the language are also encoded here. All expressions can be reordered and short-circuited (hence language semantics for short-circuiting must be encoded in its translation to IR)

This **IR** was designed to be highly general, to allow for potentially any different language frontends, and as such supports full pointer arithmetic, function calls in expressions, void calls, and all possible control flows (as a block graph). It can also optionally define an integer overflow handler.

ThreeCode

A control flow graph based **pesudo-assembly** upon which we perform all of our high-level optimizations. UNFINISHED!!!

ArmCode

A control flow graph based arm representation,

Implementation

Language of Choice

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 to .
- 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.

External Libraries

A small list of 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, we will go through them and explain our implementation:

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

0.1 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.

Project Management

Future Improvements