Project Tricarbon

aka Cyclopropatriene

Cimple Compiler Components

Design Document

Noam Preil, predraft 3, 1/3/2021

Rationale

Project Tricarbon - C_3 , or Cimple Compiler Components - is my attempt at making a cleaner compiler ecosystem. The key word involved is *components*; I have no intention of writing frontends for every language, architectural translators for every ISA, or object wrappers for every platform's pet executable format. While some degree of arrogance is of course required in order to think myself capable of designing an interface that will *allow* for such, I am not nearly stupid enough to believe that I could then fully *implement* it.

"Okay, but what does that mean? *What really even is this?"*

Basically, instead of writing a *library* that people can use for optimization and machine code generation (like, say, QBE, MIR, or libLLVM), I'm writing a simple *protocol* that various programs can use. That is, you could for instance wrap libLLVM in a small shim to use it for optimizations - but you could also construct a simple native compiler for your new language by just writing a parser that speaks Tricarbon's ASL (more on that shortly), and building a static executable with a complete parser, semantic analysis, optimizations, and code generation! The only work you need to do is to add a parser.

"Wait, don't language frontends need semantic analysis, too?"

In existing "language-independent" compiler systems, yes. Tricarbon is designed differently, in a way that shifts complexity from the frontends to tricarbon itself (the *protocol*, not even the implementation). However, I believe that the simplification that this allows for each individual component more than makes up for it, and that overall complexity decreases. Additionally, this actually simplifies other parts of tricarbon as well, so I believe it is a net gain in simplicity.

"Isn't this too ambitious? Why shouldn't I use a more practical project, like QBE or LLVM?"

If I was promising to implement support for every ISA, every operating system, every programming langage, and so on, than this would not be merely ambitious, but lethally stupid and absurdly arrogant - *especially* if I claimed I would provide that in a timely manner! The goal with tricarbon is *not* to support everything - or even all that much! The goal is to create a protocol that reduces compiler complexity, and to write a few components that use it. Anyone else can then make their new project speak the same protocol, enriching the ecosystem for everyone entangled in its... el... eternally erudite grasp? Sorry, I can’t start alliterating and then *stop*!

Okay, so I’m going to try to be a bit formal from here on out, and let the design do the talking.

Abstract

Compilation is typically viewed as an NxM problem - every language needs a frontend, plus a backend for each target. Projects like LLVM aim to reduce this by sharing backends and optimizations. This traditional view is inaccurate: the frontend for a language written to use libLLVM needs both parsing and semantic analysis, which can be broken into separate steps (and frontends do this split internally). Furthermore, the backends can be split into "machine code generation" and "object file wrapping," a distinction understood by Zig’s self-hosting compiler. Combined with the typical lock-in employed by libraries like LLVM or QBE, writing a compiler is needlessly painful today. I devise a simple protocol compilation components can use to solve both issues.

Notation

For the purpose of clarity, I use Zig’s syntax for disambiguation of arrays, slices, pointers-to-one, and pointers-to-many. Code samples are typically provided in both C99 and Zig, mostly to showcase how two different languages with two different libraries can cooperate using a standardized protocol. For types, each code block is assumed to have an implicit include of the *stddef.h* and *stdint.h* headers. Note that const is not used in the C samples, and zig code conforms to the grammar as laid out in the 0.7.1 language reference after modification with accepted proposals.

```
*T    // pointer to a single T
[*]T  // pointer to many T, length is unknown
[N]T  // array size N of T
*[N]T // pointer to array size N of T
[]T   // slice of T. length is runtime known. Where val is a slice, val.ptr is a [*]T, and val.len is the length.
```

Additionally, any of the pointer types other than pointer-to-one can be annotated with an arbitrary *sentinel*, a value which marks the end of the data.

[*:0]char *is a pointer-to-many characters, with unknown length, where the last value is a 0.*
This is identical to a C string, as the meaning is typically used.
Note that C's *char *is ambiguous between* [*]char *and* *char *in this notation!*
The notation is used specifically to disambiguate such cases.

Goals and ideals

In order to be able to work on a design and test if it meets the goals, said goals need to be explicit and well-defined.

- Freedom
- Correctness
- Simplicity
    - Architectural simplicity
    - Code simplicity
    - Data simplicity
- Ease of use / friendliness
- Language-independence
- Position-independence
- Compiler performance

Freedom

Any user of tricarbon-based components should be able to fully ignore me. This is the core design ideal, which must never be violated in any manner. For instance, simplicity is my third listed goal - if someone wants their component to be complex and to do things I personally oppose (a C++ or Rust frontend, for instance), they MUST be able to do so.

The most practical implication of this is preventing vendor lock-in. With LLVM, GCC, and even QBE, it's all or nothing - you write your piece to tightly integrate with the ecosystem in question, and it's nearly possible to switch if you want to then e.g. use your initially LLVM-based frontend with GCC's backends. With tricarbon, that's never an issue; there is no official

API or even ABI, so you can integrate with *anybody*'s backends and tooling.

Moreover, the ecosystem should be fully *decentralized.* This means that there are *no* "official" tools. Anyone could make their own list of known components, for instance. A solution to improve discoverability may be worth working out in the future.

This principle can be seen largely in the separation of components. If someone likes my x64 backend but thinks my type analyzer is trash, they can take what they like and discard what they don't. This decision can be made simply by not compiling the unwanted components! Compare this to LLVM, where everything is one big library, or Zig's stage1, which requires *every possible LLVM target* to be included in the build to meet the ideal of first-class cross-compilation and explicitly does not support building with them disabled.

This ideal is somewhat paradoxical in nature, as I'm *imposing* my axiomatic ideal of agency on all authors of tricarbon components. However, a conforming component *can* perform multiple functions, so it's still fully possible to ignore me on this and build a component that engenders lock-in (a component which targets many backends, for instance). It's always possible to give up freedoms, it's much harder to gain them back later. Succintly, the ideal is to give people the option to make the choice of freedom for themselves.

Simplicity

The third Ideal is simplicity. If anything can be done in a simpler manner without restricting freedom, it SHOULD be done as such. Note well that I said *should*, not *must*. Sometimes, making the architecture a tiny bit more complex can allow making the implementations much simpler, or faster, or so on. Such tradeoffs are typically made on a case-by-case basis. However, any case where any such tradeoff is made MUST contain the rationale in the design. Generally, architectural simplicity is more important than code and data simplicity, which are roughly equal.

Architectural simplicity

The project's architecture needs to be as simple as possible. There are many reasons for this, so a short list of the most important ones follows. It makes it easy for people with little experience in compilation to learn how it works and use it, making the project beginner-friendly. It makes it easier to extend later, adding features which are needed for new pieces. It allows for composing behavior of arbitrary complexity out of simple primitives.

Code simplicity

Implementations need to be simple. This is more relaxed; implementations can *choose* complexity if they so choose. However, tricarbon MUST *encourage* simplicity of implementation; it must beeasy to write simple code to interact with tricarbon. This, again, contributes to beginner-friendliness.

Data simplicity

Data structures need to be simple. This contributes to a positive feedback loop with code simplicity and, yet again, makes the project more beginner friendly. You shouldn't need to pore over the design to figure out how types are designed; it should be intuitive.

TODO

Notes on the rest of the ideals; focusing on the design for now.

Overview

There are a few classes of components, separated not by their functionality but by their communication patterns:

- Drivers
- Parsers
- Modifiers

- Code generators
- Postprocessors
- Object wrappers

To qualify as a component, a program MUST implement at least one of these, and MAY implement all of them. Every component MUST be executable as a binary, and MAY additionally be loadable as a dynamic object. The driver invokes other components as subprocesses, piping their inputs in and reading the outputs over a simple protocol. Note: this is currently under consideration; it is possible that the transport mechanism will be reworked shortly.

Drivers

A driver is a program that is invoked by the user, either directly or as a library. Drivers, in turn, invoke 1 parser per input file, any number of modifiers, zero or one code generator, and zero or one object wrapper. For instance, my CLI driver is invoked as follows:

```
c3 hello.zig -o hello --target=native # zig parser -> modifiers -> codegen -> ELF wrapper, on Linux
c3 hello.c -std=c99 -o hello.asl # c parser -> modifiers -> disk
c3 hello.coy -o hello.s # coyote parser -> modifiers -> codegen -> disk
```

Note that the driver is language-independent and yet understands language-specific options. I have been convinced that the original design for this is untenable; how this is achieved is currently a TODO. It is likely that the solution will involve drivers parsing their own arguments list and fully understanding them (which also allows for e.g. UI-driven drivers, as the spec will leave that stage unspecified). The trick that needs to be worked out is specifically how to pass arguments from the *drivers* to the other components; it is possible this will be accomplished through a generalized, mandated format for arguments, but the ramifications need to be worked out. Class-specific option formats may be an acceptable option, as well. Another option may be to have a specified list of known flags per-class, with a way of specifying component-specific flags.

Parsers

A parser is any program which reads in *arbitrary* inputs and produces a Stoplight Structure (henceforth referred to as an SS). Input may be a programming language, IR, machine code, or any arbitrary format. The instructions in the SS produced by a parser MAY be typed; the driver MUST assume that they are not.

Modifier

A modifier is any program which reads in an SS and produces a modified SS. For instance, typification and semantic analysis are fed the raw SS produced by the parser, and give an SS with analyzed instructions as their output. For most input languages, the driver needs to ensure that the SS is typified and analyzed immediately after parsing; this is NOT a hard requirement (e.g. when reading in LLVM IR, the driver may not need to do so).

Code generator

A code generator reads in an SS and produces non-SS output. Typically, this is assembly; it MAY be raw machine code, or C, etc.

Postprocessors

A postprocessor reads in the code generator's output and processes it. An assembler is a postprocessor, for instance.

Object wrapper

An object wrapper is given a code generator's output, after any postprocessing, wraps it in a sane manner, and outputs the result. The driver then handles e.g. writing it to disk. Object wrappers MUST gracefully error out when given invalid input.

Abstract Syntax List

The core innovation of Tricarbon is the Stoplight Structure, or the Abstract Syntax List. Unlike traditional compilers, which use a parse tree, parsers written to speak the Tricarbon protocol create a position-independent list of nodes. The reason this is so significant is that the ASL is *designed* to efficiently represent various languages, instead of attempting (as GCC did) to adapt an AST to fit various languages. The ASL itself is sent as the form of communication, but it is also a "Stoplight Structure," a concept which is explained below.

The ASL provides all three forms of simplicity through a few unique properties:

- Position independence
- Relative indices
- Stoplight structures
- Generalized nodes
- Rootlessness
- Data-as-metadata
- Embedded instructions

Many of these properties are intertwined; I suggest skimming them all once to get a basic picture, then rereading them to fill in the details.

To give an idea of code and data simplicity, here is a quick sample to showcase usage.

C99

```
#include <stdio.h>
#include "tricarbon.h"

int
main(int argc, char **argv)
{
	struct c3_asl asl;
	// Convenience function:
	// reads the ASL into two separate spots on the heap (for more intuitive memory management),
	// and provides the pointers and sizes. Note that on a system where the processes can share
	// memory, this MAY bootstrap a bidirectional pipe and use shared memory.
	if(c3_asl_read_parent(&asl) != 0)
```

```
	return 1;
	// ...
	// do optimizations here...
	return c3_asl_write(stdout);
}
```

Zig

```
const c3 = @import("tricarbon");

pub const main = fn() void {
	// choose an allocator
	const allocator = ...;
	const ASL = c3.readParent(allocator);
	for(ASL.nodes) |node| {
		...
	}
};

// ASL looks like this:
struct {
	nodes: [:0]c3.ASL.Node,
	str: [:0]u8,
}
```

Layout

The ASL layout in memory is undefined. Libraries and applications are free to choose. When sending over a pipe, it MUST be laid out as follows: spec version, as specified later, followed by the node count (64-bit), the full node list, the length of the character buffer (64-bit), and the raw character buffer (which MUST NOT be null terminated). Components SHOULD use the ASL directly, and not merely use it as an intermediary format, but they MAY choose to do so anyways.

Position-independence

A few unusual properties are used to gain position independence. See also: Stoplight Structures,

The fundamental string type of tricarbon is not, in fact, the string. Character pointers are, shockingly enough, *not* position-independent! Slices have the same problem; they still rely on a pointer. As such, the ASL's list of nodes is followed immediately by a character buffer, and

strings are a slice-like type consisting of *index* and length.

```
struct c3_node{
	enum c3_node_tag tag;
	union{
		...

	};
	size_t next;
};

struct c3_string{
	size_t index;
	size_t length;
};

struct c3_asl{
	struct c3_node *nodes;
	char *str;
	size_t node_count;
	size_t char_count;
};

bool
read_asl(struct c3_asl *asl)
{
	fread(asl->node_count...);
	asl->nodes = malloc(sizeof(struct c3_node) * asl->node_count);
	if(!asl_nodes){
		return false;
	}
	...
}
```

Relative indices

All node indices in the list are relative to the current node. See also: rootlessness. Index zero, the "self" node, always indicates termination.

Stoplight structures

The "ASL" is, in fact, *not* a list. It’s many lists, and a tree, and one list, at the same time. This sounds confusing, but is actually really straightforward. There are three main structures: a "red" tree, which is identical to a traditional AST, a "green" list, which I’ve been calling the ASL, and "yellow" chains. The "real" structure is the green list *and* the yellow chains, as nodes

contain the next link in the chain. Here’s a real example to make it more intuitive:

Let’s say we’re representing as the "root" node the following expression, in which a and b are both signed integers, and thus works equally well as Zig or C99:

```
a + b
```

This example is useful because it’s easily representible with a more traditional AST-like representation. The root node is an *expression*, which has the operator "ADDITION_UBFLOW" (in both C and Zig, overflow is UB on signed integers). It has no "next" node, as it is the root, which cannot have sibling nodes in an AST. Its desendant chain begins at an "identifier" for "a", which links to the "b" identifier, which is terminal. Neither identifier node hs children.

The green list has three nodes: "expression," "a identifier", "b identifier" at indices 0, 1, and 2 respectively. This list contains two yellow chains: the first is [0], the second contains [1,2]. There is also one red tree, the AST: it has the expression as the root, and two identifiers as children.

There are numerous reasons for this. Firstly, it allows storing ASTs easily, *and* languages which don’t built to a traditional AST, such as DragonFruit (https://github.com/limnarch). It takes a bit of thinking to grok, but is largely intuitive. It serves language-indepedence and position-independence in a simple manner. It removes the need for components to build an AST out of it, operate, and return to an ASL - that is, it promotes the AST from a protocol detail which components attempt to avoid as much as possible into a convenient tool that obviates the need for another structure. This improves code simplicity, and means heightened performance as there is no need to serialize / deserialize from the ASL. It also provides a simple mechanism for multi-file compilations, which is decribed in the section on rootlessness. There’s probably even more reasons I’m missing.

This closely matches my ideal of making simple primitives that serve many functions at once in composable ways; the usage of a spotlight structure allows for many, many distinct applications.

Generalized nodes

A node is a tagged union. A node may be a file, an expression, a statement, an IR bundle, and so on. This may be implemented differently depending on language support, as long as it correctly maps to the underlying data. One important note is that nodes cannot have more than one child. Each layer of this "red tree" is built over a singly-linked list referred to as a "yellow chain" which is embedded in the "green list" called the ASL (see the section on Stoplight Structures).

Rootlessness

See the Stoplight Structures section before reading this.

It is not entirely accurate to state that the ASL is rootless; the "green list" has no root, but the first node is also the first node of the root *chain*. This composes well with relatively indices to render multi-file compilation into a triviality: the driver invokes the parser for each file, extracts the lists, and then appends the root of each ASL to the root chain of the prior list.

Data as metadata

A "metadata note" on a node references the beginning of a yellow chain, and includes a tag indicating the meaning. This allows an optimizer to persist the original instruction list as metadata, giving more information to backends! Optimizers can, if they all persist the original instruction metadata when present, fully optimize instruction lists while preserving the full original state of a function. This also means further optimization passes have more information to work off of!

Embedded instructions

Unlike a tradtional compiler pipeline, in which the AST is converted into IR for analysis, optimization, and then