Project Tricarbon

aka Cyclopropatriene

Cimple Compiler Components

Design Document

Noam Preil, predraft 4, 1/3/2021

Acknowledgements

In no particular order, a huge thanks to DarkUranium, floatcomplex, and Eleanor for explaining exactly how I was being stupid and how to stop. Any quality found in the tricarbon spec is because of them, any mistakes are my fault :)

Rationale

Project Tricarbon - C_3 , "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 TIRE (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 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 - with good reason - 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, outside of explicitly labeled code samples, I use Zig's syntax for disambiguation of arrays, slices, pointers-to-one, and pointers-to-many, as it is more intuitive than C's syntax; this is not a dig at C. Full code samples are typically provided in both C99 and Zig. For types, each C99 code block is assumed to have an implicit include of the *stddef.h* and *stdint.h* headers; any other headers will be included explicitly. 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.
```

Note that slices can be viewed as structures akin to the following C:

```
struct slice{
	T* ptr; // pointer to many
```

```
        size_t len;
};
```

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.

For more details, see the Zig language reference. This section is provided to make the rest of the document intuitive to non-Zig speakers.

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
- Platform-independence
- 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.

Correctness

Generally, if there's an easy way to do something, and a correct way, the protocol must choose the correct method. This is kept as objective as is possible: the rules for "What is correct?" reduce down to "Does it fit the other stated ideals?" There is no intent on sticking to dogma here. For instance, if a change allows heightened performance without compromising any ideals, it MUST be taken. If a change simplifies the architecture, but restricts flexibility, it MUST NOT be taken. This shouldn’t need to be written out explicitly, but I’ve done so anyways largely as a reminder to myself to look past my own biases and to avoid falling into the trap of the sunk cost fallacy.

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.

Ease of use

The tricarbon protocol should be very easy to use. An experienced programmer with no prior knowledge of tricarbon should be able to write a frontend for a simple language within a week. This is served by and very tightly related to simplicity, but is a separate, explicit Ideal. Ease of use guides the focus of simplicity; if two choices are equally simple, or one simplifies the data at the expense of the code, ease of use is the tiebreaker.

An otherwise implicit aspect of the emphasis on beginner friendliness is high-quality documentation. The protocol itself, in addition to the exacting details of a spec, MUST be explained concisely in a way that people with *no prior experience* with compilation can fully understand it quickly. This requirement is NOT imposed on components. Getting people with low prior exposure to programming itself up to speed rapidly is out of scope of the project; using tricarbon components as the basis on an in-depth exploration of programming may be a worthwhile endeavour, but is not germane to the protocol.

Language independence

Tricarbon must not depend on *any* language to any real degree. Language samples are given in C and Zig; neither is given special favor by the protocol. I use C primarily, as it is my language of choice; I use Zig as it simplifies other code samples. The protocol itself MUST NOT be biased towards them.

Platform independence

Tricarbon must not be tied to existing operating systems, architectures, or ecosystems.

Performance

Without violating the higher ideals, performance MUST be maximized. However, it is the least important ideal: every other ideal listed takes precedence over performance. Modern CPUs can perform more than ten *billion* operations per core per second, and that’s only increasing, but no magic hardware will make projects easier to use, or fix bugs. To quote the Plan 9 coding guide: "Don’t optimize unless you’ve measured the code and it is too slow. Fix the data structures and the algorithms instead of going for little 5% tunings. If a segment of code is taking 100 nanoseconds during the full compilation process, and you can reduce this to 1 nanosecond, don’t bother. Those 99 nanoseconds don’t matter; focus on bottlenecks, not inefficiencies.

Overview

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

- Drivers
- Parsers
- Analyzer
- 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 able to be configured at compile-time to be built into a static library, and MAY additionally be executable as a standalone binary and / or loadable as a dynamic object. That is, it MUST be possible to build static libraries for each component, and merge them with the driver's main() in order to produce a single compiler toolchain from the components, and components MAY choose to be designed to allow run-time configuration instead.

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
```

How the driver understands and passes flags to other components is left unspecified. The transport mechanism used to pass TIREs to components is left unspecified.

Parsers

A parser is any program which reads in *arbitrary* inputs and produces a TIRE. Input may be a programming language, IR, machine code, or any arbitrary format. The instructions in the TIRE produced by a parser MAY be typed. Drivers are responsible for knowing whether or not analysis is needed.

Modifier

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

Analyzer

An analyzer is a special subclass of modifier. Its input is possibly-typed, its output MUST be typed. Exactly one analyzer MUST be used in each compilation pipeline unless the driver is certain it is unneeded (for a customized JIT, for instance). Generally, a language-specific tricarbon-based compiler toolchain should only need to know about the existence of one analyzer. Analyzers are specified explicitly in order to avoid having two formats for the TIRE, one untyped and one typed, which all components must understand. With the presence of analyzers, the only components which need to deal with untyped instructions are analyzers; all others receive their inputs after analysis.

Note that, since some languages require compile-time execution for typification, analyzers intended to be language-independent need to be able to deal with this. To avoid mandating such complexity on compilers for simple languages, analyzers MAY be language-specific. It is the driver's job to know what analyzer it can use for what language, and whether a given analyzer is language-independent. How this is accomplished is left unspecified; it may be a command-line driver flag, or a configuration file, or any other mechanism. Additionally, if a component other than an analyzer sees unanalyzed instructions, it MUST gracefully error out, and MUST NOT crash.

Code generator

A code generator reads in an TIRE and produces non-TIRE 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.

Tricarbon Interchange Representation Edifice

The protocol of Tricarbon is built over the Tricarbon Interchange Representation Edifice, or TIRE. This name was chosen despite being stupid because it does not need to be changed as the format is iterated over and the details change - and because it avoids any unintended connotations from other names. For instance, the previously chosen name of ’Abstract Syntax List’ (or ASL) looks like a typo of "AST," and implies that the data should be viewed as a list, which is not the case.

TIRE is *designed* to efficiently represent various languages, instead of attempting (as GCC did) to adapt an AST to fit various languages. TIRE is meant exclusively as an interchange format; simple transformations should be possible on it, but more complex operations will likely need to treat the TIRE as immutable and to build a new one from scratch.

The TIRE provides all three forms of simplicity through the emergent behavior of a few properties:

- Position independence
    - Relative indices
    - Statically sized nodes
- Binary tree
- 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_tire TIRE;
	// Convenience function:
	// reads the TIRE 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(&TIRE) != 0)
		return 1;
	// ...
	// do optimizations here...
	return c3_tire_write(stdout);
}
```

Zig

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

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

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

Layout

The TIRE 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 MAY use an

TIRE directly, but are not required to do so; they should do whatever makes the most sense.

Position-independence

The fundamental string type of tricarbon is not a traditional C string. Character pointers are, shockingly enough, *not* position-independent! Slices have the same problem; they still rely on a pointer. As such, the TIRE'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 left;
	size_t right;
};

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. Index zero, the "self" node, always indicates termination.

Statically sized 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. Every node may have exactly two children, forming a binary tree; either or both may be marked as terminal.

Data as metadata

A "metadata node" 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! Additionally, this allows high-level "template instructions," which are explained in the *Embedded Instructions* section.

Embedded instructions

Unlike a tradtional compiler pipeline, in which the AST is distinct from the IR, the TIRE contains the IR within the tree. Components MAY preserve raw statements as well. Combined with arbitrary nodes as metadata, this allows for high-level template instructions, such as method invocation or trait application, to be encoded directly within the tree, while linking to their lowered forms for components which do not recognize them.

More information will be added here in the future; for now, the existing architecture needs to be tested in code :)