Project Tricarbon

aka Cyclopropatriene

Cimple Compiler Components

Design Document

Noam Preil, predraft 5, 2021-01-12

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 :)

Abstract

Project Tricarbon - C_3 , "Cimple Compiler Components" - is my attempt at standardization of compilation. The key word involved is *components*; instead of a *library* for optimization and machine code generation (like, say, QBE, MIR, or libLLVM), Tricarbon is a simple *protocol*. All of the listed libraries employ lock-in by their very nature: attempting to port a GCC frontend to speak LLVM is practically impossible, as is porting an LLVM backend to work with QBE's optimizers instead of LLVM's, etc. Additionally, creating any of those components is needlessly complex, since all three libraries are built over "pure" SSA-ed IR which, to make matters worse, limits how precisely the language can be encoded, making it impossible to write simple static analyzers to work with these formats. Tricarbon solves the issues of lock-in, complexity, and over-abstraction.

Goals and ideals

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

- Ecosystemic freedom
- Simplicity
    - Architectural simplicity
    - Code simplicity
    - Data simplicity
- Reliability
- Ease of use / friendliness
- Language-independence
- Platform-independence
- Performance

Ecosystemic freedom

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 impossible to switch if you want to then e.g. use your initially LLVM-based frontend with GCC’s backends, or you disapprove of GCC’s direction and want to switch to QBE. With tricarbon, that’s never an issue; you can integrate with *anybody*’s backends and tooling, and switch seamlessly. Note that this doesn’t mean the ability to leave the *protocol* - but by limiting tricarbon’s scope to usage as a protocol, the lock-in is shifted to the protocol instead of to specific tools.

Simplicity

The second Ideal is simplicity. All three forms of simplicity listed below must be prioritized above everything except for correctness and freedom. Simplicity is considered as a *resource*; all functionality necessarily increases complexity, and any feature MUST be demonstrated to be worth the complexity it incurs in order to be accpeted, and the simplicity "expended" on it. Sometimes, making the architecture a 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 be justified explicitly in the design. Additionally, there is an upper limit to how much complexity will be accepted.

Architectural simplicity

The project’s architecture needs to be as simple as possible. There are many reasons for this. It makes it easy for people with little experience in compilation to learn how it works and use it, making the project more beginner-friendly. It encourages composition of behavior of arbitrary complexity out of simple primitives, improving the functionality-per-simplicity ratio. It makes it easier to keep the code and data simple, as well.

Code simplicity

Implementations can *choose* complexity if they so choose, but tricarbon MUST *encourage* simplicity of implementation; it MUST be easy to write simple code to interact with tricarbon without needing a library. If an implementation chooses to add additional complexity internally, that is allowed, but it MUST not affect its usage of the protocol. This, again, contributes to beginner-friendliness. It also makes it easier to write a component without poring over the documentation constantly. Furthermore, it allows code to be maintainable and future-proof.

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, for instance; it should be intuitive.

Reliability

Tricarbon components must be reliable. A component that compiles today should compile 30 years from now with a correct compiler without needing any upkeep, and it should interact with new components flawlessly. This means that the protocol needs to be *finalized*: once it is finished, that's it. There will not be any further revisions to the protocol, ever; the only acceptable changes are clarifying ambiguous wording and further refining the specification, not extending it. As such, it will not be released until it is 100% complete and has been tested thoroughly in varying use cases. The usage of tree "stems" should ameliorate the need for extensions, and the ability to represent raw IR makes the *worst* case for tricarbon equivalent to fitting a language to QBE or LLVM.

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 such as Brainfuck within a week. This is served by and very tightly related to simplicity, but is a sepa-

rate, explicit Ideal. Ease of use largely guides the focus of simplicity; if two choices are equally simple, or one simplifies the data at the expense of an equal increase in complexity to 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 - but with a strong background in systems programming - can fully understand it quickly. Getting people with low prior exposure to programming itself up to speed rapidly is a worthwhile endeavour, but is entirely out of scope of the project.

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. The protocol must be easy to interact with in any reasonable language (where reasonable excludes, for instance, esolangs like brainfuck; this is again not a dogmatic stance). Furthermore, the tree design must accomodate as much as is possible without violating the ideal of simplicity - in practice, this means the usage of stems and low-level language "primitives" such as continuations which are designed to represent many higher-level concepts. Continuations for instance are used as a primitive on which to build generators, coroutines, exceptions, and more, allowing multiple distinct concepts from varied languages to be efficiently encoded.

Platform independence

Tricarbon must not be tied to existing operating systems, architectures, or ecosystems. Tricarbon’s nature as a protocol largely renders this moot, though a few considerations - such as endianness - remain. This is the driving motivator for leaving some behaviors unspecified - if the protocol was limited to POSIX, for instance, pipes would likely be a mandated method of communication.

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 code more readable, projects easier to use, or bugs easier to fix. 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." In that vein, the design of the TIRE needs to be carefully thought out and tested for performance.

Overview

There are a few classes of components, separated by their communication patterns:

- Drivers
- Readers
- Modifiers
- Writers

To qualify as a component, a tool MUST implement at least one component type. Every component MUST be configurable 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 directly, whether via a CLI or e.g. an IDE's language support plugin. Drivers, in turn, invoke 1 parser per input file, any needed analyzers, any number of modifiers, and a code generator. Drivers MAY run additional custom

passes at any point.

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. These are both out of scope of the protocol. This may be pipes, temporary files, shared memory, or any other mechanism.

Reader

A reader 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 reader MAY be typed. The reader MUST annotate the TIRE with metadata indicating what forms of analysis are required (typification, validation, etc); predefined metadata nodes or custom stems may be used. If a driver does not recognize a stem, it MUST error out. Analyzers are simply modifiers that the compiler knows to use for analysis.

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.

Writer

A writer reads in an TIRE and produces non-TIRE output. Typically, this is assembly; it MAY be raw machine code, or C, etc. This is the final step that Tricarbon concerns itself with. A driver MAY run additional passes; this is left unspecified.

Tricarbon Internal Representation Exchange

The protocol of Tricarbon is built over the Tricarbon Internal Representation Exchange, or TIRE. The 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 inter-change 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 is designed to meet every one of the Ideals through the emergent behavior of a few properties:

- Index-based strings
- Relative indices
- Statically sized nodes
- Chained metadata
- Stems
- Embedded instructions

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. Components MAY use an TIRE directly, but are not required to do so; they should do whatever makes the most sense. When sending a TIRE, components MUST lay it out as follows: node count (32-bit), node list, character count (32-bit), string buffer. Components MUST NOT insert padding below the transport layer.

Index-based strings

The fundamental string type of tricarbon is not character pointer, or a slice; neither are position-independent. Instead, the TIRE's list of nodes is followed immediately by a character buffer, and strings are nodes consisting of *index* and length.

Relative indices

All node indices in the list are relative to the current node. Index zero, the "self" node, always indicates termination. This makes nodes position-independent - so long as all transiently referenced nodes are preserved in the correct positions, a node remains valid. This allows indices to sanely be 16-bit, as nodes should never be more than 37 *thousand* entries away by direct refer-

ence - which makes 64-bit nodes possible.

Binary tree with statically sized nodes

Nodes consist of a tag, and three indices. 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 has exactly two children, forming a binary tree; either or both may be marked as terminal. Every node also has an index for metadata. As a result, every node is exactly as big as three relative indices and the tag - currently, with a 16-bit tag and three 16-bit relative indices, each node is exactly eight bytes. The TIRE is a binary tree whose root node is the first entry in the list of nodes.

Data as metadata

Metadata nodes are simply nodes which were accessed via a metadata index. They made encode various pieces of information, and components MUST assume that metadata is meaningful, and cannot be dropped - optimizers MUST preserve metadata on preserved nodes, for instance, and persist portions not known to be meaningless.

Embedded instructions

Unlike a tradtional compiler pipeline, in which the AST is distinct from the IR, the TIRE embeds the IR within the tree. Components MAY preserve raw statements as well. Combined with the usage of arbitrary nodes as metadata, this allows for high-level "stem" 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.

Stems

"Stems" are special nodes marked with the "stem" tag. A stem encodes a high-level meaning, such as "invoke method" or "call exceptional routine," while encoding as metadata primitive language-independent instructions. For instance, a language implementing exceptions can define a thread-local exception "register," and have lowered instructions which store to it on exception, and utilize the primitive continuation support to implement exceptions. Stems for "exceptional call" encode simultaneously the semantic meaning and the raw IR, allowing at once tools which need concrete syntax trees and tools which needs raw instructions to handle the same tree.

History and rationales

What follows is a list of changes and reasoning for decisions which doesn't make sense to include in the relevant contexts. Formatting of this section is a bit off; not a priority.

- Predraft 5

  - Replaces "Spotlight Structures" with simple binary tree. Reason: eliminates complexity and ambiguity.

  - Removed postprocessors and object wrappers. Reason: needless complexity - Tricarbon need not care what happens after codegen.

  - Parsers must notate TIREs with needed analysis. The driver is responsible for recognizing this metadata.

    - Rejected: analyzers needing to report their known stems and the driver querying this. Reason: needless complexity.

    - Resultant improvements:

  - Removed comments about decentralized ecosystem. Reason: this is not yet decided, further contemplation and discussion is required.

    - Current thoughts: an official ecosystem might be fine so long as it's not, *in practice*, required - if the "pick-n-choose" philosophy works here, it's probably fine. The important question is practicality; if people gravitate towards a specific ecosystem and it engenders lock-in, that's as problematic as if the protocol endorsed it.

- Renamed templates to stems. Reason: disambiguation - removes risk of confusion with C++ concept and resultant association with generics.

- Reduce node size to 64-bit. Reason: improves performance - faster allocations, reduced RAM usage. Cost: nodes can only reference nodes within $2 pow 15$ indices of themselves.

- Renamed ASL to TIRE. Reason: disambiguation and longevity. The previously chosen name of ’Abstract Syntax List’ - ASL - looks like a typo of "AST," and further implies that the data should be viewed as a list, which is inaccurate. TIRE will not need to be changed as the design matures.

- Remove sections of the document which weren’t needed.

- Rename parser to reader. Reason: can read in non-programmatic inputs, such as asm or IR.

- Rename code generator to writer. Reason: can write out non-machine-code outputs.

- Removes general ideal of freedom. Reason: impractical and irrelevant. Tricarbon is a protocol which by its nature entrenches its ideals in its own decisions, and has no affect on components. Replaced with ecosystemic freedom, which is the sole part which is both meaningful and practical.