Project Tricarbon

aka Cyclopropatriene

Cimple Compiler Components

Design Document

Noam Preil, Predraft 8, Sat Feb 13 18:29:35 EST 2021

First public v1 candidate

Acknowledgements

In no particular order, a huge thanks to DarkUranium, floatcomplex, Eleanor, and ikskuh for explaining exactly how I was being stupid and how to stop. Any quality found herein is their fault, I take credit only for the mistakes. Each of them has provided invaluable feedback which changed the very nature of the project. Thanks, fellow humans! Gratitude goes out additionally to g_w1 for helping with the zyg compiler and with ironing out last few details of this draft. Thanks as well to Andrew Kelley and Drew DeVault for inspiring some of the details, and for helping to build my interest in langdev and compiler programming with Ziglang and $redacted respectively.

Abstract

Project Tricarbon - C_3 , "Cimple Compiler Components" - is two projects. First, it is a project to design a protocol by which various compilation components can speak. Second, it is the project to implement a few compilers using the protocol. 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* intended to solve lock-in and over-abstraction.

Ideals

These are the specific qualities the specification is intended to embody, rendered explicit in order to better facilitate testing the design conceptually. As a direct result of these ideals, a number of use cases were dropped, and the project's scope was reduced significantly.

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

Realism

In essence, "better the flawed project today than the perfect project next century." Instead of some pipe dream that promises to solve every possible use case eventually, Tricarbon will be a narrow project which solves *some* uses cases *now* - where "now" means "in a few months plus testing." The key impact so far has been to drop non-compilation use cases from the initial protocol. Maybe a future version will accomodate such, if a way to do so while still meeting the ideals of the project is determined.

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 accepted, 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.

As a direct result of this ideal, roughly two thirds of the originally intended functionality has been dropped completely. While a number of practical reasons to prioritize simplicity in its many forms are listed, the core reason is that simple software is better than complex software.

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 makes it easier to keep the code and data simple, as well. One key facet of this is that Tricarbon doesn’t try to specify every possible behavior. It specifies exactly as much as it needs to in order to be useful and meet its goals, no

more, and no less. There are numerous places where the specification leaves behavior unspecified, and the precise details can reasonably vary between implementations without impacting the protocol's goals, or they are irrelevant to the protocol.

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 improves readability and maintainability.

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. This also contributes positively to 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.

It is worth noting that if the only implementation that ever pops up is my own, it is a form of lock-in. However, the goal is to make better solutions *practical*, not to mandate them; if centralization around a specific ecosystem occurs, that is acceptable, so long as it is still realistic

to leave.

Also worth noting is that it is impossible to *fully* meet this goal, since the core design of all components will always be affected by the decision to use tricarbon - however, multiple tricarbon ecosystems can realistically coexist and interact. In keeping with the Ideal of realism, it is important to push this goal as much as is possible without pretending that we can fully achieve it.

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. Ideally, this would mean only releasing the specification after it was 100% tested and completely ready. The usage of tree "stems" should ameliorate the need for extensions. However, that is not realistic.

Inevitably, some things *will* have to change. Pretending otherwise does not solve any issues. As such, the protocol is versioned. All components MUST gracefully error out when a TIRE from an unknown version is encountered. All components SHOULD support outputs produced by older versions, but they may choose not to. It is expected that updates will be few and far between and have minimal impact, but it is important that they be handled well when they do happen.

Since most changes should be minimal, TIREs encode *two* versions: first, the target version of the protocol, and second, the oldest known version which supports it. This makes it possible for an older component to see that it will not have any issues with a tree produced by a newer component.

Modifiers which handle multiple versions as inputs SHOULD retain compatibility with whichever version they received, but they MAY emit a tree which uses new features. For the sake of being explicit, modifiers MAY also emit a TIRE with an *older* target version.

No space is set aside in the version space for "is this a draft?" The first draft will be v0, the second draft will be v1, and so on. If the first official release comes after Draft 16, it will be v16.

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 separate, 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.

Note as well that not all languages are simple, and it is not the goal of the protocol to make complex tasks simple. Compiling *simple* languages should be simple. Furthermore, not all programmers have systems programming experience. To that end, documentation is provided, but that is the extend of what can reasonably be done.

Language independence

Tricarbon must minimize dependence on specific languages. The protocol must be easy to interact with in any reasonable language - a clause meant to exclude esolangs, such as Brainfuck, and languages not intended for systems programming, like POSIX shell. While some degree of dependence is unavoidable, tricarbon’s design is to be tested against a wide variety of languages to maximize compatibility with wildly different paradigms.

This also serves reliability: components written in Amazing New Language #17 should interact well with components that haven’t been maintained for a few years.

Platform independence

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

There are a few items which will not be considered however. For instance, tricarbon assumes that bytes are octets - 8 bits - and makes no accomodation for, for instance, 7-bit machines. Tricarbon should not be used on such machines.

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 - Tricarbon's fundamental data structure - needs to be carefully thought out and tested for performance. Designing the TIRE for performance means a reduced incentive to seek out minor tunings, improving code simplicity.

Impacts

To accomodate the various goals, a few constraints have been needed. In pursuit of simplicity and reliability, the scope of this specification has been limited significantly. For the sake of simplicity and performance, the data structure has been completely redesigned twice.

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. Other methods of usage may be supported, but are outside the scope of this specification.

Drivers

A driver is a program that is invoked by the user's Intent, whether via a CLI or e.g. an IDE's language support plugin. Drivers, in turn, invoke readers to parse input files, any analyzers required, any number of modifiers, and optionally a writer. The behavior after modification is unspecified - drivers may execute a code generator for the full output, one per input file, or anything else entirely.

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. TIREs may be passed over pipes, temporary files, shared memory, or any other mechanism. This is intended to allow for flexibility of implementation. For single-threaded drivers which execute components in the main thread, components may simply pass the TIRE on the stack and / or the heap. Components MUST offer functions with the ABIs specified in their respective sections, using the TIRE encoding outlined below.

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. For the sake of clarity, this explicitly includes a TIRE; it is fully valid for a reader to simply read a TIRE from disk or over a pipe, for instance.

Readers MUST expose a single function using the C ABI with this signature:

```
int(char *, TIRE*)
```

That function must return one (1) on success and zero (0) on failure. The first argument must be an absolute path (resolved by the driver) containing the file to read. The second argument must be the address of a memory location to which to write the produced TIRE. The reader must NOT write to the address containing the path.

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 which is instrumented with additional instructions. Since modifiers run after analysis, they MAY ignore stems, or they may be made with awareness of specific language features to optimize compilation or outputs, or to improve behavior. Language- or semantic- specific modifiers MAY require stems to be present.

Modifiers must expose a function with the following ABI:

```
int(TIRE*, TIRE*)
```

As with the reader, the function must return one (1) on success and zero(0) on failure. The first argument is the input TIRE, and the second is the output. If the two arguments are equal, the modifier MUST take care not to clobber its input before using it. If necessary, it MAY copy the input internally.

Writer

A writer reads in an TIRE and produces arbitrary output. Typically, this is assembly; it MAY be raw machine code, or C, etc.

Writers must expose a function with the ABI

```
int(TIRE*, char *)
```

This functionwhich, again, returns 1 on success and 0 on failure. The first argument is the tire to write, and the second is the path to which to write it. The path should either be "-" to indicate standard output, NULL to indicate that the writer should annotate the output as a string in the TIRE and append it to the root node, or an absolute path to which it should be written.

Tricarbon Internal Representation Encoding

The protocol of Tricarbon is built over the Tricarbon Internal Representation Encoding, or TIRE. The TIRE is a tree structure with embedded IR. TIRE is meant primarily 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 is designed to meet every one of the Ideals through the emergent behavior of a few innovative properties, as well as some ideas borrowed from other projects, with a dash of conventional understanding.

Layout

The TIRE layout while in use 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. TIREs in transit over the specified static function ABI MUST be laid out as indicated in the pseudocode below. Names needs not match those used here. TIREs passed around via other mechanisms, such as pipes, can use any format they want - which explicitly allows for compression via e.g. LZ4.

Note that if desired, a component may expose other static functions, which can return the TIRE in other formats.

```
packed structure tire {
    // lowest version of the protocol the tire complies with. For this draft, this MUST be set to zero.
    u32 version_min
    // highest version of the protocol the tire complies with. For this draft, this MUST be set to zero.
    u32 version_max
    // Number of nodes - read 4 bytes * node_count to fill in the nodes array.
    // Note that this includes children! If there is one node with five children, this will read as eleven!
    u32 node_count
    // The nodes themselves. Entries are either nodes, or indices pointing to other nodes.
    // This is an array, sized at node_count * 4. Actual nodes take two entries.
    u32 nodes[]
    // Number of strings - read 8 bytes * string_count to fill in the strings array
    u32 string_count
    // The strings themselves. Strings are encoded as indices into the chars array plus lengths.
    // index-based slices, effectively.
    packed structure string {
```

```
        u32 index
                u32 len
        } strings[]
        // The number of characters
        // Read this number of bytes after char_count itself to populate the chars array
        u32 char_count
        // The raw character buffer. Strings in this buffer are NOT zero terminated.
        // Components MAY zero-terminate them, but MUST NOT assume that they are zero-terminated.
        char chars[]
        // The number of tokens. Read 6 * token_count to fill in the tokens array
        u32 token_count
        packed structure token{
                // The type of token. Some types are reserved by tricarbon and mandated for all tires, but most
                // tokens are language - and thus toolchain - specific
                u16 tag
                // the handle of a string in strings containing the token contents
                u32 string
        } tokens[]
}
```

Nodes look like this:

```
packed struct node{
        // The node type. For nodes representing input tokens, this matches
        // tire.tokens[index].tag
        u16 tag
        // The number of children of this node
        u16 kid_count
        // Either the index in tire.tokens to find the first token of this node, or
        // the index of the first child.
        u32 index
}
```

The TIRE represents a tree built over a list of nodes. Every TIRE MUST contain a root node. The node tag indicates which kind of node is encoded. kid_count contains the number of children of the given node. A node's children are included directly after it in the nodes array. Each child is a 32-bit unsigned integer representing an absolute index of the child in the nodes array.

There are two general types of nodes. First, there are nodes which represent input tokens directly. For these nodes, node.tag is equal to tire.tokens[node.index].tag. Second are nodes encoding compiler details, such as IR. For these, index is the index of the first child, if kid_count >= 1.

Considered but rejected: 16-bit relative indices with extension nodes for far-away nodes. This would reduce memory usage to 6+2N in the common case (from the current 8+4N), in exchange for an addition 6 bytes for far-away nodes. It would in all likelihood reduce RAM by >10%, and improve performance, but at the cost of simplicity. The 8+4N scheme was devised as a replacement for a system which had roughly 16+10N on average, and this scheme successfully improves performance significantly while *also* improving simplicity. The 6+2N scheme would be an improvement compared to the old system, but 8+4N provides excellent performance *without* compromising on simplicity.

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 stems to encode the semantic meaning. Stems are used practicality everywhere at the parsing stage, but most components will ignore them in favor of IR after analysis.

A stem encodes the semantic meaning of the instruction alongside a lowered representation, melding a concrete syntax tree with SSA IR. The meaning of a stem is encoded via two strings: the namespace and the id. Language-specific extensions MUST encode the canonical name of the language as a prefix to the namespace’s name. Extension strings may be defined by arbitrary extensions of this specification. Stems MUST always lower to other nodes, which are typically IR.

No stems are defined in this specification. All stems MUST be defined in language or toolchain extensions.

Tokens

Tokens consist of a 16-bit tag, plus a 32-bit string handle. The string can then be found in tire.strings[string_handle], which gives the index into tire.chars and the length. Note that tokens in tire.strings are NOT sequential. To poke left and right for neighboring tokens, components MUST poke in tire.tokens itself.

Tags

While some tags are exclusive to tokens, and some are exclusive to nodes, *no* reuse is permitted.

Token tags

Whitespace = 0

Common tags

Comment = 1
Contextual = 2
Type = 3
Identifier = 4

Node tags

Stem = 5
Export = 6
Visibility = 7
Block = 8

Whitespace

Whitespace tokens do not show up in the node list. They include spaces, tabs, carriage returns, and newlines.

Comments

Comments appear as nodes. This is to facilitate, for instance, semantic analysis of commented-out code by static analyers. After the first parse failure within a comment, no more nodes will be added. Comment tokens refer only to the comments themselves. Any descendants of a comment node have their token index replaced with a string handle.

Contextuals

A contextual token is one whose meaning is ambiguous - these are intended for use in contextual grammars. Contextual *nodes* are not yet defined.

Types

A type token is any token representing a type. This can be ‘bool‘ in C, for instance, ‘noreturn‘ in zig, and so on. Something like ‘struct foo‘ in C gets two tokens: a language-specific struct token, and an identifier token.

Identifiers

An identifier is any token representing a user-provided identifier.

Stems

Every stem must have at least one child, which is a special node which does not fit the typical structure. It contains only two string handles. The first is the language or toolchain identifier, and the second is a stem identifier. Imports in the zyg compiler have "zyg" "import" for instance, which is typically written as zig.import. The reason a dedicated child is used for both strings instead of using two is to allow easier interning and comparison: by deduplicating these stem types, a compiler can make stem comparisons into an integer comparison. The second child of a stem must be either zero to indicate absence or the lowered form of the stem, if applicable. Any additional children are defined by the toolchain or by a language consensus.

Note: the burden of ensuring no conflicting stems lies on component authors - especially driver authors. Pick a meaningful name for the designer - for instance, the name of the project ("zyg"), the name of the language if you have consensus among multiple compiler authors ("zig"). It might make sense to prepend "draft-" or your name for stems whose design is incomplete. Fortunately, it does not matter if components use the same name for stems as long as the

components are not interacting - the worst that happens if an incompatibility between two components.

If such a case does come up, however, it is the responsibility of the involved parties to work out an equitable solution. This is a social problem, not a technical one.

Exports

Export nodes must contain exactly four children. First, the ABI. Typically, this is "c". Secondly, the exported name. For instance, if you have ‘@export("foo", bar)‘ in zig code, this is "foo". Third, the type of the exported value, and fourth the value itself.

Other

All other node types are left undefined for this draft. It is expected that large swathes of this specification will be discarded for the second draft, having been informed by the first components written against this draft. More details will be added then.