Project Tricarbon

aka Cyclopropatriene

Cimple Compiler Components

Design Document

Noam Preil, predraft 7, 2021-01-27

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.

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. Furthermore, the IRs used by all mainstream compilers discard a lot of information. Tricarbon is intended to solve lock-in 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. These goals have had numerous profound impacts on the design, the biggest of which is to narrow the scope in favor of simplicity and reliability.

Goals

These are broad goals of the project.

- Promote language-independent tooling
- Promote more powerful analysis tools
    - Source-level optimization assistant - "32-bit variable ‘x‘ is bounded in [3, 224], narrow to 8-bit?"
- Make developing custom high-quality compilers less time-intensive

Ideals

These are the specific qualities the specification is intended to embody.

- Realism
- Ecosystemic freedom

- Simplicity
    - Architectural simplicity
    - Code simplicity
    - Data simplicity
- 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."

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. The protocol is also simple enough that a self-contained compiler could reasonably stop speaking it easily, avoiding lock-in even to tricarbon itself.

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.

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, more than half of the originally intended functionality has been dropped completely.

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 readabilityand 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.

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 handle unknown versions. 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.

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.

Language independence

Tricarbon must minimize dependence on specific languages. Language samples are given in C, but C is not given special favor by the protocol. 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.

Platform independence

Tricarbon must not be 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.

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.

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 reader per input file, any analyzers required, and any number of modifiers. The behavior after modification is unspecified - drivers may execute a code generator for the full output, one per input file, or something 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. This may be pipes, temporary files, shared memory, or any other mechanism. This is intended to allow for flexibility of implementation.

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. 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.

Considered, but rejected, was the idea of requiring the reader to invoke analysis itself; however, there are two issues with this.

- It requires the reader to know about other components. With the current design, the reader only needs to mark down which forms of analysis are needed, not the tools which execute them, and it never needs to invoke anything else. This means that the only component which ever requires knowledge of other components is the driver, allowing for a cleaner separation of components.

- One of the goals of Tricarbon is to promote language-independent tooling; if the parsers and analyzers were more tightly bound together, it would encourage a greater degree of language-dependence.

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. All modifiers MUST be able to work with TIREs as defined by this specification. Modifiers MAY additionally support stems.

Writer

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

Tricarbon Internal Representation Exchange

The protocol of Tricarbon is built over the Tricarbon Internal Representation Exchange, or TIRE. The TIRE is a tree structure with embedded IR. 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 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
- Embedded instructions
- Stems

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. Note that since nodes are statically sized, the node list’s size is known the instant the node count is read, and so the character count is at a known position, as well.

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 reference - which makes each node a static 64-bit value, or eight bytes.

External references go through a dedicated node, which combines its left and right indices into a single signed 32-bit value, allowing for external references to access nodes which are further away, while still allowing internal nodes to be a mere eight bytes.

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.

Chained metadata

Metadata nodes are simply nodes which were accessed via a metadata index. Metadata is defined by the Tricarbon spec just as other nodes. Metadata nodes are chained by attaching further nodes as the metadata node, chaining until a terminal node is encountered.

Embedded instructions

Unlike a tradtional compiler pipeline, in which the AST is distinct from the IR, the TIRE embeds the IR within the tree. 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 stems to encode the semantic meaning.

Note that stems are effectively just annotated IR instructions. 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 IR. The lowered form may contain further stems, but recursive resolution MUST yield IR.

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

Tree structure

The TIRE encodes objects which can have arbitrary numbers of children, but each node can only have two children, as it is a binary tree. To allow encoding, for instance, multiple statements per function, any object X of tag Y, where both its lhs and rhs children have tag Y, unless otherwise specified in this document, the lhs and rhs should be viewed as direct children of the parent of X. As a simple example, if a function's lhs node is a statement, and its lhs and rhs are statements, the tree should be viewed as if the two statements were directly attached to the function. This is a recursive feature; an arbitrary number of children can be encoded simply by tacking more chains on. Children are found depth-first.

The rest of the structure is not yet defined; more testing in code is required.

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 7
    - Discard primitives entirely. Tricarbon core is just IR in the shape of a tree.
    - Significantly narrow project scope.
- 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.