C++17
Compile Time
Register Machines

Daniel Nikpayuk

December 13, 2021

Abstract

The intention of this essay is to introduce a collection of specifications for implementing a system of C++17 compile time register machines. By writing out the details in an essay style format it is also the intention here to provide key strategies for narrowing down possible implementations further, as well as offering a narrative story in the hopes of aiding later on in both proof-verification and debugging. It is assumed the reader has a reasonable understanding of automata theory and finite state machines.

This essay's specification—which will be elaborated upon in the philosophy section below—can be broadly categorized by the following design considerations:

1. **Theory**
   (a) **Space Design**: We will be designing a space whose objects are register machine programs.
   (b) **Algebra Design**: We will design such programs to be atomic, constructive, and callable.
2. **Practice**
   (a) **Hardware Constraints**: We will consider the most relevant hardware bottleneck designs.
   (b) **Software Constraints**: We will consider the most relevant software bottleneck designs.
   (c) **Community Constraints**: We will consider designs which ideally satisfy user-friendliness.

Philosophy

To take a step back from our *specification as object*, we can restate the main goal of this essay here as that of telling a narrative story which will eventually help us to implement a system of register machines using C++17's toolset. [1]

Unfortunately things are not as simple as that, or at least this is the philosophy I'm designing from: For me, such a system of Turing complete register machines suffers from complexity, which means no single narrative story is sufficient to describe all the patterns of interest. My compromise (and belief) is that we in fact need two narratives to tell the design wanting to be told.

The philosophy of this essay can then be said to be a narrative stitched together from two others, which are themselves informed by three canonical stories.

[1] Implementing in C++17 is in fact the original application of this design.

Story 1: A Humanities Perspective

The first story in navigating a system of register machines is to conceptualize it using a humanist inspired triple: [2]

$$\{\,\text{text, reader, hermeneutic}\,\}$$

How does this conceptualization work?

We'll start with the `reader`: To put it most directly, register machines—as finite state machines—are expected to be equipped with memory devices known as registers, but from our above humanist perspective we will anthropomorphize this to in fact be our `reader`. Why? Reading is a passive act, [3] which for us suggests that whatever is read is expected to change the internal nature of the reader. This is no less true for a memory of registers to which we apply their respective state machines accordingly, and for which we expect those registers to be mutated as they "read" their given programs.

With this orientation, we can now associate our idea of a `text` with register machine programs—though often we tend to focus more specifically on program *controllers* when no confusion arises. [4] This is reasonable as texts are fixed and unchanging, as are programs, and both are read to create changes in their respective readers.

All that's left to conceptualize in our triple then is the `hermeneutic`, which is our given reading of the text. How do we qualify this? In our register machine framework this would be our state machines. Why? The state machine (the interpretation) we're applying takes both the `reader` and the `text` and creates an interaction between them. The idea is that the application of the hermeneutic to the reader and the current location of the text changes the reader (as well as the current location of the text), but this is exactly what state machines and their transition functions do.

To summarize, a system of register machines is decomposed as follows:

- **text**: corresponds to programs, often with controllers in mind.
- **reader**: corresponds to the registers.
- **hermeneutic**: corresponds to the finite state machines (transition functions?) that act on both controller instructions as well as the registers to return the updated register states, as well as the next location within the program.

Finally, if we were to take this analogy to its logical conclusion, we would say that a reading of a text is a process where the reader goes from location to location, applies its chosen hermeneutics changing itself along the way, then moving on to the next location to repeat. Such interactions continue until the reading of the text is considered successful and complete. As such a process coincides with how register machines work to perform their respective computations, the intuition is confirmed (or at least not denied).

With this story now told, I would like to add that the main purpose in framing register machine systems from a humanities lens is in fact to clarify the roles of the actors in this otherwise complex play.

Story 2: A Equivalence Subplot

Now that we know the main characters, let's reorient and focus on a major subplot, one which itself will ultimately help us understand the first of the two narratives being presented.

This plot comes from theoretical computing science and automata theory, and is all about ensuring our register machines are Turing complete. The idea is since our reader—our memory of registers—takes the same shape regardless of texts or hermeneutics, we can hold it fixed and abstract it out. From here, we can then focus on telling the story of the relationship between such texts and hermeneutics, or rather between programs and finite state machines.

In particular we are interested in the *correspondence* between programs and finite state machines that we will call `evaluators`. If we defined our programs from a mathematical lens as a *space*, we would ask what the nature of

[2] I suspect most readers (of this essay) are already familiar with what *texts* and *readers* are within natural language contexts, but the idea of *hermeneutics* might be less well known: Mostly it's the idea that given a text, it can have more than one reading based on how you interpret it, and so there becomes a need to study the logic of possible interpretations of texts more generally.

[3] This is true even when a reader is associated with the idea of having *agency*.

[4] The reason we don't associate texts directly with program controllers is that texts also have additional information we tend to take for granted but for which we otherwise shouldn't actually forget about—such as where we start, or where we currently are in our reading, etc.

all such *objects* inhabiting this space could even be? We would also want to ask if our space is an *algebra*, but let's focus on the first question first: The historical consensus is that a given object belonging to this space is a program if and only if it is a "list of instructions" that we can actually compute. Putting this another way, we would say the object that is our program has an above mentioned *evaluator*.

At this point we don't need to know the specifics of what evaluators are to say that for each program in our space of programs we can consider the fact that there is a corresponding evaluator in a space of evaluators. Conceptually this is a nice clean plot point, but it's not a very interesting one, or even a useful story: A theory of computation isn't all that meaningful if we have to manually (with cleverness and originality) construct an unique evaluator for each program we want computed.

The plot moves forward by asking if we can do better: Can we find a single "meta-evaluator" such that it is finitely described and can itself simulate all other evaluators in our infinite evaluator space? Spoilers: The answer to this is yes, but with some tradeoffs. Either way it is known as a universal Turing machine.

With our correspondence now described, we still have the second question to answer regarding our space of programs: Is our space an algebra? It's easy enough to determine a catenation operator (a monoid) to construct composite programs from both atomic and other composite programs, but does this algebraic quality extend to the space of corresponding evaluators? Does it translate to our meta-evaluator? The answer here is yes and also yes.

In any case, these questions and answers about our space of programs are our second story.

Story 3: A Memory Arc

Our second story abstracted out the memory from our system, so we will want a third story now to discuss it in detail.

Our memory space is a collection of registers for which we have random access to both read and write their contents. Unlike our math lens though, the theory of computation doesn't shy away from practical considerations such as performance. This third story then is a story of the practical, and in particular the bottlenecks that are most commonly associated with memory access.

In terms of practical memory access, the thing to realize here is that whatever our system ends up being it is intended to be compile time computable, and will thus be simulated on top of our compiler's own computations. Given this, we can already expect two constraints:

- First, it is our expectation that our system will inherit any and all of the practical hardware constraints that our compiler must itself consider—the most notable one for us being *finite memory*.
- Secondly, even if we were to design our register machine programs performantly, we are still running overhead costs that add up at the compiler level. In particular, simulated read/write operations, as well as simulated program calls—given their frequency of use within evaluators—should be prioritized for mitigation.

Beyond that, we also have what might be called auxiliary memory access concerns—performance concerns not directly encoded in either the hardware or software, but decided on by the community of users. For the most part such value-systems cannot be determined in advance as they extend out to include our cultures, politics, and other contextual content, but there are some low-level policies of access or even user-friendliness that can still be predicted.

The Narrative

With these substories now in motion, we have enough backstory that we can finally organize our narrative:

1. **Space**: Our system of register machines assumes **Turing completeness**.

   We must identify the C++17 constructs we intend to use to implement our reader, our texts, and our hermeneutics, and further identify how to build a corresponding meta-hermeneutic (evaluator).

2. **Algebra**: Our system of register machines assumes **constructivity** and **callability** of programs.

   This means that we should not only be able to build composite programs out of atomic programs (constructivity), but we should also be able to create composite programs out of other composite programs (callability).

3. **Hardware**: Our system of register machines assumes **memory limits**.

   In terms of C++17, this will often translate as *nesting depth* limits for us. Such limits will be mitigated using the `trampolining paradigm`.

4. **Software**: Our system of register machines assumes **reasonable performance**.

   We will need to mitigate register read/write access using what's called the `blocking paradigm`. As for mitigating program calls we will prioritize internal function template calls over tail calls.

5. **Community**: Our system of register machines assumes a **user-friendly interface** for architects to write, debug, and run their own compile time programs.

   This will be achieved using the `detour paradigm` which will allow us to hide housekeeping instructions—abstracting them away from general architect programs.

If you're wondering about certain details introduced in this narrative which weren't discussed in the canonical stories above, such details will now be elaborated upon.

Methodology

We first need to discuss the idea of a `vehicle of transmission`.

As mentioned in the narrative we are simulating our register machine system on top of the compiler's own computations, and as such we need to identify the mechanism or medium—our vehicle—in which our compile time computational information will be transmitted.

Foregoing the suspense, *function templates* are our vehicle.

Space Methods

Ultimately the goal here is to simulate a system of register machines using functions and function templates satisfying the C++17 standard toolset.

Turing Completeness

Given access to C++ variadic packs, we can successfully simulate a Turing complete register machine system based off the theoretical (automata theory) result that says a finite state machine with two stacks as its memory system is sufficient to be Turing complete.

State Machines

Atomic Machines

With this in mind, the following provides a baseline anatomy for how we will implement such finite state machines, making note that two main variadic packs are used to implement our theoretical stacks:

Stack 1:	`controls` {d, m, n, c, i, j},				`registers...`
Stack 2:	`heap 0, heap 1,`	`heap 2, heap 3,`	`heap 4,`	`heap 5,`	`arguments...`
	Stage 1 (register mutations)	**Stage 2** (function calls)	(constants)	(save/restore)	

Compound Machines

It's not so much that there are "compound" machines themselves, rather it's that the predefined atomic machines are monadically composed in predefined ways (as programs) through specified *controllers*. This follows the traditional design of register machines more generally. Specifically the major design is borrowed from and quite

closely copying "Structure and Interpretation of Computer Programs" Chapter 5, which describes a standalone register machine system implemented in the Scheme programming language, a variant within the larger LISP style of languages.

Algebraic Methods

Continuation Passing

We know in advance we will be using function templates to implement these compile time machines. As such, we need a *vehicle* to take the current state and pass it to the next state (with associated instruction) so as it act on it next.

A most natural design then is to use a continuation passing monad with enough complexity that we achieve Turing completeness as an emergent effect.

Program Calls

As mentioned above, we start with atomic machines but we do not actually build compound machines out of them, so it is better to reframe the description as *programs*. As such, we start with atomic programs and then build compound programs from them. Such compound programs correspond to a chaining of atomic machines to start, but in the long run it is also advantageous to be able to *call* programs as if they were atomic machines.

We do this to satisfy the user-friendly design principle known as *modularity* of design.

Hardware Methods

although in the context of a compiler and our **vehicle of transmission** this generally means *nesting depth* constraints.

Nesting Depths

To restate the second design constraint here:

> "Assumes finite memory (at any given time, but is potentially extensible)."

Another way to put this—given our reliance on function templates as our vehicle of compile time computation, another more accurate way to phrase this is as:

> Assumes a finite/fixed nesting depth for function calls (at any given time).

Within the methodology, we have enough restrictions (details) now to choose the method for mitigating finite nesting depths: *Trampolining*.

Intersectionality of design. Beyond the theoretical, this design constraint is actually most important because it needs to be considered within every other practical design, which is to say it is at the intersection of all other designs.

Software Methods

It is my understanding the following two concerns:

- caching memory for faster access
- preferencing inlining over function calls

are two of the major concerns of compiler optimizers. Although I am not an expert myself in this area, these two concerns do inform my software methods here.

Reasonable Performance

As far as software performance goes, the thing to remember is that we're simulating computations on top of an existing software computation—the compiler. At the same time, in general our simulation is that of a compiler, or rather an assembler itself. Keeping this is mind, we can take lessons learned from compiler optimizations themselves.

First is *inlining*, but that requires greater support for manipulating programs as objects, while also maintaining their types, so that when they are continuation passed the native compiler will do its work without throwing errors. Unfortunately I myself haven't fully succeeded in this area of research, and have for this reason honestly abandoned it in favor of other approaches.

Secondly then, is to optimize against each individual hermeneutic machine, but as there are only finitely many such machines this approach doesn't scale. With that said, we can without loss of generality assume that each individual hermeneutic machine is of reasonable performance, in which case it then becomes a matter of finding bottlenecks in terms of how these machines are used—distributional patterns of use.

From this perspective, program calls, especially recursive calls would be the notable constraint. I've read many social media accounts from practicing compiler theorists, I will claim that this observation also aligns with traditional wisdom when it comes to compiler optimization theory as well.

With a goal in mind, how do we design for program call optimization then?

Secondly, in terms of compiler bottlenecks, the major one to consider is peformance costs for when our register machines make recursive (program) calls. As we are simulating on top of the compiler, this cost adds up more quickly than the rest. Given this, special care must be taken to minimize the cost of simulated recursion, ideally even to make use of the compiler's own recursive mechanisms without much in the way of our own overhead. As it turns out this is possible, and is why we privilege *internal function template calls* over *tail function template calls*.

Tail call vs Internal Call

The other major performance bottleneck to consider as mentioned earlier is that of accessing registers within a two stack memory system.

As for program calls themselves: Given the two stack memory design, there is a minimum core of programs and hermeneutic machines required to achieve near-random memory access. These fall into the *genre* of programs (texts) known as *block* programs/machines.

Community Methods

Unfortunatey it is a side effect of our vehicle of transmission that under the current design the architect is required to add housekeeping instructions to their code that are otherwise tedious an uninformative as to the intended nature of their program. To abstract this away, we need an additional *community* mechanism to detour to specific housekeeping machines while simultaneously preserving the current indices and our ability to return to the existing navigational path, furthermore without interfering with our ability to trampoline.

Detour Abstraction

Proofs

Anatomy of a Compile Time Register Machine (C++ Function Template)

Each **atomic machine** has the following form:

```
template<>
struct machine<name>
{
      template<stack...>
      static constexpr auto result(heaps...) {...}
}
```

The *name* allows for dispatching (template resolution), while the *constexpr function* has a single *stack* made up of a variadic pack symbolically representing *registers*, along with a fixed number of *heaps* which are also made up of variadic packs, but which are *cached*, and thus more expensive in general.

We then build **compound machines** by chaining them together with a **controller**. Such machines and controllers are organized into a **hierarchy** of machine orders using a *monadic* narrative design: The idea is that the atomics of a higher level are constructed from the compounds of lower levels which—assuming self-similarity propogates throughout—then allows this pattern to scale:

Atomic Programs

Compound Programs

The idea is that with the chains of machines (at a given level) we can either end the chain with a *halting instruction* or a *passing instruction*. Halters effectively become standalone functions (with some interface hiding the chain), returning some standalone value. Passers on the other hand are intended to continuation pass to other machines at higher orders.

This then implies a few consequences for the design of each individual machine:

- Each machine is required to carry its own controller, which includes required indices (as well as a nesting depth counter), along with index iterators. Peformance and modularization design suggests such info should generally be carried on the stack.
- Each machine that has a higher order is required to carry the controller, indices, iterators, of the machine it is eventually returning to. Abstraction-wise it makes the most sense to carry this info in a designated heap.

Trampolining

Internal Function Calls

A model for CTRM Trampolining which uses *internal* rather than tail function calls

```
outer call machine

                    current call machine

                                          inner call machine
```

Program Calls

Block Programs

Linear Programs

User Programs

filler.