C++17
Compile Time
Register Machines

Daniel Nikpayuk

December 13, 2021

Abstract

The intention of this essay is to introduce a specification along with key strategies 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 a narrative story in the hopes of aiding later on in proof-verification, as well as general troubleshooting (debugging) following any actual implementation. It is assumed the reader already has a reasonable understanding of automata theory and finite state machines.

The design considerations of the expected specification—which will be elaborated upon in the philosophy section below—are categorized as follows:

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

Ultimately the goal here is to tell a narrative story that will eventually help us to verify a system of register machines implemented using the C++17 specification toolset.

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

Story 1: A Humanist Perspective

The first way navigate a system of register machines is to conceptualize them as a humanist inspired triple:

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

I suspect most readers (of this essay) are already familiar with what a text is, and what a reader is, but the idea of *hermeneutics* is a little less known. Mostly it's the idea that any given text 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.

How then do we conceptualize systems of register machines this way?

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 memory to be our reader. Why? Reading is a passive act, even if a reader is associated with the idea of having *agency*: The expectation is that whatever is read will change the internal nature of the reader. This is no less true of a memory of registers which are expected to be mutated as they read their given programs, applying their state machines' applications accordingly.

Thus, we have programs as our texts—often we tend to focus more specifically on the program *controllers* though—and in turn we're left with our state machines corresponding with our hermeneutic, our given reading of the text. Why? The state machine, or correspondingly the hermeneutic/interpretation we're applying takes both the text and reader, and creates an interaction between them. Our text being made up of instructions which hold valuable fixed content but are otherwise largely symbolic, while our reader already holding its own relevant content when combined with the current location of text and hermeneutic modify the reader. The interaction continues this way, and so on and so on until the computation halts, and the reading of the text is considered successful.

This is a pretty lofty narrative, so let's summarize:

- **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 state.

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

Story 2: An Equivalent Retelling

Now that we know the main characters of our story, let's reorient and focus on the plot—which itself will ultimately help us understand the overarching narrative being presented in what follows.

The 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 memory of registers, our reader, takes the same shape regardless of texts or hermeneutics, we can hold it fixed. We'll make it our protagonist, and abstract it out. From here, we can then focus on telling the relationship between the text and the hermeneutic, or rather the programs and the finite state machines of interest.

In particular we are interested in the correspondence between programs and what we will call *evaluators*. If we view our programs as forming a space, then we would recognize a given object as a program belonging to this space only if it had an evaluator. This is to say, a "list of instructions" is only a program if we can actually evaluate or compute it.

In turn, what this says is that for each program in our space of programs, 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 or even 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 then by asking if we can do better? Can we find a single (finitely described) "meta-evaluator" such that it can itself simulate all other evaluators in our evaluator space?

Spoilers: The answer to this is yes, and it's known as a universal Turing machine.

With this then being the plot, we have enough backstory that we can now organize our narrative design using the following categories:

Category 1: Space

Must be compile time Turing complete (at least extensibly so)

Category 2: Algebra

Assumes constructivity of programs, where one starts with atomic programs and uses them to create compound programs

Assumes callability of programs, where one can additionally create compound programs out of either atomic programs or other compound programs

Category 3: Hardware

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

It should first be noted that our simulation—our compile time computation—is expected to run on top of one's compiler. As such, we inherit any of the bottleneck constraints that the compiler must consider. The most notable one being finite memory constraints, although in the context of a compiler and our vehicle of transmission this generally means *nesting depth* constraints.

Category 4: Software

Assumes reasonable performance for program calls, especially recursive program calls

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*.

Category 5: Community

Assumes a user-friendly interface for architects to write, debug, and run their own compile time programs

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.

Methodology

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

This means we need to first consider whether this is even theoretically possible, and if it is, we need to follow up in asking if it's also practical. In particular: What are the bottlenecks we need to mitigate? Can we mitigate those bottlenecks? How best can we mitigate them?

The too long don't read answer is that it is in fact both theoretically and practically possible: The purpose of this essay then is to build a clean, minimal, theoretical narrative (a story to orient our reading of the system) along with proofs demonstrating the initial design constraints are satisfied accordingly.

Function templates are our vehicle of transmission. What are we transmitting? The compile time computation.

Category 1: Space

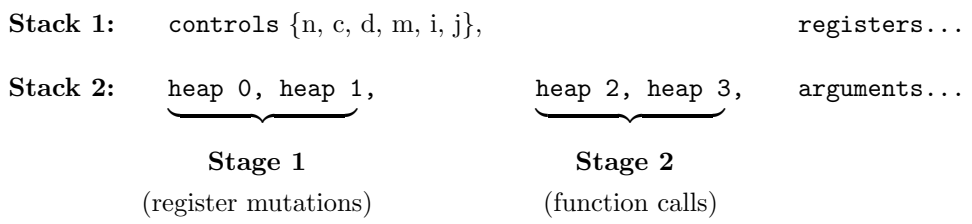
Turing Completion

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:



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.

Category 2: Algebra

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 completion 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.

Category 3: Hardware

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.

Category 4: Software

Reasonable Performance

Tail call vs Internal Call.

Category 5: Community

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.