C++17
Compile Time
Register Machines

Daniel Nikpayuk

December 13, 2021

Abstract

Philosophy

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.

As for the initial design constraints, I list them casually, intuitively, here as follows:

1. **Potential**: Must be compile time Turing complete (at least extensibly so).

2. **Expressivity**: Assumes constructivity of programs, where one starts with atomic programs and uses them to create compound programs.

3. **Expressivity**: Assumes callability of programs, where one can additionally create compound programs out of either atomic programs or other compound programs.

4. **Bottleneck**: Assumes finite memory (at any given time, but is potentially extensible).

5. **Bottleneck**: Assumes reasonable performance for program calls, especially recursive program calls.

6. **Bottleneck**: Assumes a user-friendly interface for architects to write, debug, and run their own compile time programs.

Methodology

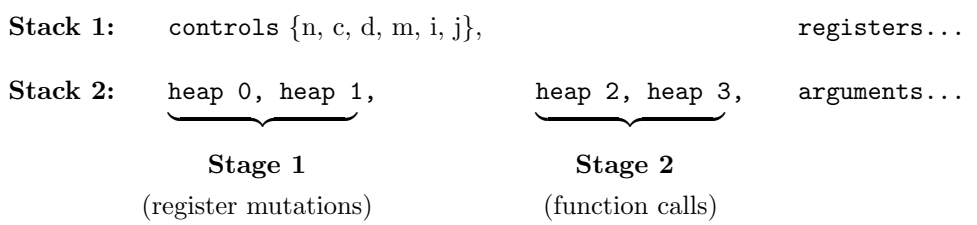
Turing Completion

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.

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.

Reasonable Performance

Tail call vs Internal Call.

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.

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.